Programmer's Reference Guide

c-tree Plus™

c-tree Plus™ File Handler

Programmer's Reference Guide

Sixth Edition

For c-tree Plus™ Version 6.0 or later,
the FairCom® Server,
and
the FairCom® SQL Server

©1984, 1986, 1987, 1988, 1990 FairCom Corporation

ALL RIGHTS RESERVED

Published by

FairCom Corporation
4006 West Broadway
Columbia, MO 65203
(314) 445-6833

Copyright © 1984, 1986, 1987, 1988, 1989, 1990 FairCom Corporation

All rights reserved. No part of this publication may be stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of FairCom Corporation. Printed in the United States of America

First Printing: October 1990

c-tree, c-tree Plus, r-tree, d-tree, the c-tree circular disk logo, and FairCom are trademarks of FairCom Corporation.

UNIX is a trademark of AT&T

MS-DOS, OS/2 and XENIX are trademarks of Microsoft

Macintosh is a trademark licensed to Apple Computer Co.

IBM is a trademark of International Business Machines Corp.

Dedication

In loving memory of George Fairman, Florence Stone and Rae Karsh:
special people who made a difference.

WLF

c-tree Plus™ Programmer's Reference Guide
Table of Contents

1. Introduction

What is c-tree Plus™? 1-1
c-tree Plus™ Modes 1-2
FairCom Server Multi User 1-2
FairCom Server Single User 1-3
Non-Server Multi User 1-3
Non-Server Single User 1-3
Sections of this Manual 1-4
FairCom Products 1-4
c-tree Plus™ 1-4
FairCom® Server 1-5
FairCom® SQL Server 1-5
r-tree® 1-5
d-tree™ 1-5
Natural Query™ 1-5

Basic c-tree

2. Using c-tree Plus™

Integrating c-tree Plus™ with your program 2-1
Add #include Files 2-1
Use the c-tree Plus™ Functions 2-2
Link with c-tree Plus™ Libraries 2-2
Example Programs 2-2

3. c-tree Concepts

"C" Concepts 3-1
Data Type Conventions 3-1
Strings 3-2
Prototyping 3-3
c-tree Plus™ Keys 3-4
Alternative Key Types 3-4
Fixed Length Keys 3-5
Leading Character Compression Keys 3-5
Padding Compression Keys 3-6

Combined Compression Keys 3-6
ISAM Keys 3-6
Duplicate Keys 3-6
Sequence Numbers 3-7
Data Records 3-7
Buffer Regions 3-7
Data Record Delete Flag 3-7
Resource Records in Fixed Length Files 3-8
Data and Index Files 3-8
File Modes 3-8
Virtual File Open 3-8
Fixed versus Variable Length Records 3-9
Multi-User File Mode 3-10
Transaction mode 3-10
I/O Management 3-10
Resources 3-10
Superfiles 3-10
filmod Values 3-11
Data and Index File Numbering 3-11
Low Level Functions 3-11
ISAM functions 3-12
Error Handling 3-13
File Recovery 3-13

4. Using Low Level Functions

A Simple Application 4-1
Application Outline 4-1
The Application 4-1
Initialize c-tree Plus™ 4-1
Add a record 4-2
Find a record 4-3
Delete a record 4-3
Close the system 4-3
Low Level Function Overview 4-3
System Functions 4-4
File Manipulation Functions 4-4
Data Record Manipulation 4-5
Key Manipulation 4-5
FairCom Server Transactions 4-6
Resource Functions 4-7

5. ISAM functions

A Simple Application 5-1
Application Outline 5-1
ISAM Parameters 5-1
The Application 5-2
Initialize c-tree Plus™ 5-2
Add a record 5-2

Find a record 5-2
Delete a record 5-3
Close the system 5-3
ISAM Error Handling 5-3
Current ISAM Record 5-3
ISAM Function Overview 5-4
ISAM Parameter File System Functions 5-4
Incremental ISAM System Functions 5-5
Data File and Index Functions 5-6
Set Functions. 5-7
ISAM Parameters 5-8
ISAM File Relationships 5-9
ISAM Parameter File 5-10
ISAM Parameter File Organization 5-10
Parameter File Contents 5-11
Initialization Record 5-11
Data File Description Record 5-11
Index File Description Record 5-12
Optional Index Member Record 5-14
Key Segment Description Record 5-14
Key Segment Modes 5-16
Preparing Key for Key Searches 5-18
Sequence Number Segments 5-18
Descending Key Segment Values 5-18
Alternative Collating Sequence 5-19
Database Examples 5-20
Fixed Length Parameter File Examples 5-20
Variable Length Parameter File Example 5-22
Multiple Data File Parameter Setup 5-23
Key Value Assembly 5-23
TransformKey 5-24
CurrentISAMKey 5-24
Incremental ISAM Structures 5-24
IFIL Structure 5-25
IIDX Structure 5-26
ISEG Structure 5-27
Incremental ISAM Examples 5-27
Incremental ISAM and Resources 5-29

6. Multi User Concepts

File Locking 6-1
Multi-User File Mode 6-1
Record Locking 6-1
ISAM LEVEL Data Record Locks 6-2
Using LockData with LockISAM 6-3
LockISAM and Resources 6-3
Low Level Data Record Locks 6-3
Multi-User Interference 6-4

Multi-User Updates 6-5

7. Sample Programs

Simple Applications 7-1
Simple File Maintenance 7-2
Initialize the system 7-3
Error Handling 7-4
Adding Data 7-4
Delete a key 7-6
Close the system 7-6
Same Application with ISAM functions 7-6
ISAM parameter file 7-6
Error handling 7-8
Adding Data 7-8
Delete record 7-8
Close the system 7-9
Using Incremental ISAM structures 7-9
Incremental ISAM structure 7-9
Initialize the system 7-10
File Searches and Sets 7-11
Listing a file 7-11
Sets 7-11
Scanning, Updating, Deleting 7-12
Start the scan 7-12
Scanning the file 7-13
Update the record. 7-13
Delete a record 7-13
Advanced c-tree Sample Programs 7-14
ctexmc.c 7-14
ctexmg.c 7-14
ctvxmg.c 7-14
ctixmg.c 7-15
Additional Samples 7-15

8. Advanced c-tree Concepts

Resources 8-1
Resources and Files 8-1
Resource Identification 8-2
Disabling Resources 8-3
Resource Functions 8-3
Resource Example 8-5
FairCom Defined Resources 8-5
Alternate Collating Sequence 8-6
File Security 8-6
Padding and Delimiter Values 8-6
DODA and Record Schema 8-6
IFIL information 8-6
Superfiles 8-6

Creating/Opening Superfiles and Members 8-7
Superfile Members 8-7
Closing Superfiles and Members 8-7
Restrictions when using Superfiles 8-8
Batched Operations 8-8
DoBatch parameters 8-8
Partial Key Request structure 8-9
First Call 8-9
Verify Option 8-10
Result Option 8-10
Lock Option 8-10
Completion option 8-11
Continuation Calls 8-11
Complete or Cancel Call 8-11

Non-Server Multi User

9. Non Server Multi User

Node Locks 9-1
Data Record Lock Implementations 9-2
Dummy Lock File 9-2
Dummy File Alternative 9-4

FairCom Server

10. File Server

Server-Application Relationship 10-1
Logging On and Off the Server 10-2
File Modes 10-2
Record Locking mode 10-2
Transaction mode 10-2
I/O Management 10-3
Transaction Processing 10-3
File Numbering Relaxation 10-4

11. Transaction Processing

Why do we need Transaction Processing? 11-1
What the FairCom Server Provides 11-2
Basic Transaction Processing 11-2
Create Files 11-3
Begin Transactions 11-3
End Transaction 11-3
File Modes 11-3
Record Locking 11-4
Begin 11-4
Commit 11-4
Marking Transactions 11-5

Begin 11-5
Commit 11-5
Aborting a Transaction 11-6
Savepoints 11-6
Errors in Transactions 11-7
Automatic Recovery 11-7

12. Security

Accessing File Security 12-1
Forms of System Security 12-2
Users, Groups, and Passwords 12-2
File Permissions 12-2
File Passwords 12-4

FairCom SQL Server

13. SQL Introduction

What is SQL? 13-1
FairCom SQL Server 13-1
Interactive vs. Embedded 13-1

14. FairCom SQL Server Concepts

Tables and Views 14-1
AUTHORIZATION Identifiers 14-1
Transaction Processing 14-2
Error Codes 14-3

15. Using SQL with c-tree Plus™

Initializing c-tree Plus™ 15-1
ANSI Embedded SQL C Program 15-1
SQL Declaration section 15-2
SQL Statements 15-2
Example Embedded SQL Program 15-3
Additional Information 15-4
C Function Call Interface 15-4
Output Modes 15-6

Appendices / Index

A. c-tree Plus™ Error Codes

User Errors A-1
Internal Fatal Errors A-13

B. SQL Error Codes

Standard Errors B-1
Internal Errors B-7

C. Function Name Cross Reference

Full Names C-1
Abbreviated Names C-5

D. Record Schemas

Record Schema D-1
Schema Map D-1
Schema Name Details D-2
Record Structure D-2
Field Types D-3
Create Record Schema with PutDODA D-4
Key Segments D-6

E. Common Entry Point Functions

Extended Feature Parameter Blocks E-1
Logon Block E-2
ISAM Block E-2
IFIL Block E-3
Create File Block E-3
Open File Block E-4
Key Estimate Block E-4
Function Calls E-4

Combined Index

1. Introduction

What is c-tree Plus™?

c-tree Plus is a sophisticated file management product designed to be used in a variety of operating system environments. Written in the C language, it is designed to be integrated with your application program written in C to provide you with easy to use, yet flexible data file manipulation and indexing.

The significant features of c-tree Plus are:

- **fast file access** via indices using the most advanced B + tree techniques, enhanced by our proprietary caching and compression algorithms;
- the ability to **exceed operating system limitations** on the number of files that can be open simultaneously;
- programs can be written that are **portable** over a wide variety of operating systems;
- both **single user and multi user file access**, with full file and record locking control that often exceeds the capabilities provided by the operating environment;
- **superior performance** on network and multi-user systems using the **multi-threaded FairCom Server approach to file management - a powerful Client/Server** approach to data management;
- advanced **On-Line Transaction Processing** (OLTP) and **fully automatic recovery**, including full commit and rollback, intermediate savepoints, and complete logging;
- database manipulation using standard **SQL** function calls (when using the **FairCom SQL Server**);
- a file **security** feature to prevent unauthorized access to your data files and indices;
- integration with our **r-tree®** Report Generator and **d-tree™** Development Toolbox products for a complete high level product development environment.

This Guide serves both as a introduction to **c-tree Plus**, for the new user, and as a reference for the experienced user. Please note that whenever we talk about **c-tree** we are referring to **c-tree Plus**.

c-tree Plus™ Modes

c-tree Plus can be configured to operate in a variety of ways, which we shall refer to in this manual as **modes**. The features that vary from one mode to another are single versus multi user, support of a file server, support of higher level functions such as transaction processing and SQL, and availability of source code.

This section provides you with a brief overview of the available modes. How these modes relate to the products that you can purchase, and to this manual, will be discussed in the following section.

	Multiple Users	Single User
Server	FairCom Server Multi User Mode	FairCom Server Single User Mode
Non-Server	Traditional Multi User Mode	Traditional Single User Mode

FairCom Server Multi User

In this mode you can develop programs that allow multiple users to simultaneously access your data files. A separate program, the **FairCom Server**, maintains complete control over the database. It is the only program that will read from or write to the data files and indices. The application program will communicate with the FairCom Server to gain access to the database. Performance is optimized by the use of data buffers and sophisticated use of your I/O channels. In many operating environments the FairCom Server also gives the programmer a more powerful set of file and record locking commands than is native to the environment. In distributed processor environments, such as a network, the bulk of the data access work can be done on the processor that has the best access to the database, limiting the volume of information that must be transmitted over the network.

In a multi-tasking system, such as UNIX or XENIX, the FairCom Server runs in the background. In a LAN (Local Area Network), the **FairCom Server** either runs on the same processor as the network server, or it runs on a node of the network. If it is on a node, then it may run in the background (which permits other programs to use the same node) or in the foreground (which dedicates the node to the FairCom Server). This multi-threaded design allows the Server to utilize all the previously unused time spent waiting for locks or I/O operations.

While these features are an important part of c-tree Plus (and have been available to users of c-tree™ for years), the FairCom Server goes beyond this to provide you with the features that you should expect from advanced Client/Server technology. c-tree Plus provides industrial quality On-Line Transaction Processing (OLTP) and fully automatic recovery, including full commit and rollback, intermediate savepoints, and complete logging.

A special version of the Server, the **FairCom SQL Server**, is available to provide advanced SQL access to databases managed by the FairCom Server via **interactive** and/or **embedded SQL** calls.

Source code is supplied only for the client portion of c-tree Plus, which is the portion that you link with your application. The Server itself, which contains the actual file manipulation routines, is supplied as a run-time executable file. Special licensing arrangements are necessary to distribute the executable server.

FairCom Server Single User

If you are writing applications where it is not necessary for multiple people to access the files simultaneously, you can create programs in the FairCom Server single user mode. In this case you will bind the Server to your application to create a single executable file. There will be no need for sophisticated inter-prgram communications. All of the features of the FairCom Server (or the FairCom SQL Server) are available to your application, with the exception of the multi user features. As above, source code is not provided for the server portion of c-tree Plus.

Non-Server Multi User

c-tree Plus also supports traditional **multi user file access**, where multiple programs or users can have *direct* access to the database simultaneously, with file and record locking under control of the application programmer. Each program accessing the database will maintain it's own buffer area, and will read from and write to the files and indices directly. This approach relies on the operating system for file buffering. All c-tree Plus capabilities are provided except for Transaction Processing, SQL Support and file security control. Full source code is available. This mode can be used in any operating environment that supports multi user file access and record locking.

Typically you will only be using this mode if you want to use an operating environment that is not supported by the FairCom Server. Since file buffering is minimized, performance may not match what can be achieved with the FairCom Server. The only advantage of this approach over the File Server mode is that this mode of operation is available on virtually any machine with record locking facilities; no File Server communications support is required. **Full source code is available**, which allows the user to move the product to any operating environment that supports the C language.

Non-Server Single User

Applications that are designed to have only one user accessing files at a time can use **non-server single user mode**. All of the capabilities of **c-tree Plus** are supported, with the exception of Transaction Processing, SQL Support, and file security control. The c-tree Plus code will be linked as a part of your application. The programmer can specify the amount of memory to be used for file buffering, which

can have a significant affect on the speed of the system. **Full source code is available,** which allows the user to move the product to any operating environment that supports the C language.

Sections of this Manual

This manual is divided into several major sections, which relate to the different approaches you can take to developing a product using c-tree Plus. The sections are:

- **Basic c-tree**: This section describes the basic c-tree Plus functions, both low level and ISAM, and how they are to be used in an application. Several sample programs (provided on disk as well) are used to illustrate the concepts. Multi user concepts are introduced here as well. The information provided here applies to all modes of c-tree Plus. However, if you are going to use only embedded SQL functions you only need to skim this section.
- **Non-Server Multi User**: This section applies if you are going to create a multi user application that does not use a Server. It provides details about multi user access without using the FairCom Server.
- **FairCom Server**: If you are going to use the FairCom Server you will need to study the material in this section after you have mastered the basic c-tree Plus concepts.

  The advanced c-tree Plus functions that are supported only if you are using the FairCom Server, such as Transaction processing, are described in this section as well.
- **FairCom SQL Server**: This section describes how to create a program that uses the FairCom SQL Server. These functions can be used in conjunction with other c-tree Plus functions, or you can build and manage your database with these functions alone.

A companion manual, the **c-tree Plus Function Reference Guide,** contains sections that list each of the c-tree Plus and SQL functions.

FairCom Products

c-tree Plus is just one part of a family of products offered by FairCom Corporation. These products, when used together, provide a complete range of product development tools.

c-tree Plus™

Our powerful data management product, supplying both low level and ISAM access to your data files in single user, multi user, and file server modes. A developers copy of the FairCom SQL Server is included.

FairCom® Server

If you are going to build a product for distribution that uses the FairCom Server, you will purchase an executable server for each of your customers.

FairCom® SQL Server

If you are going to build a product for distribution that uses the FairCom SQL Server, you will purchase an executable SQL server for each of your customers. This includes all of the functions that are provided with the standard FairCom Server.

r-tree®

r-tree is a powerful Report Generation tool that works in conjunction with c-tree Plus. Once you have created your files and have built a Data Dictionary that describes the files and their indices, you can create reports with r-tree by building simple scripts. These scripts can be distributed to your customers so that they can modify them, or they can be "compiled" into a form that cannot be altered by the customer.

d-tree™

d-tree is a powerful Development Environment that aids you in the development of your C language products. In includes Prototype generation, dynamic resource swapping, screen management, file restructuring, data dictionary maintenance, menu management, and more.

Natural Query™

Natural Query is a "natural language" front end that can be used to build access c-tree Plus files. In addition, it provides interactive SQL access to the c-tree Plus files if you are using the FairCom SQL Server.

Basic c-tree

This section describes the basic c-tree Plus functions, both low level and ISAM, and how they are to be used in an application. Several sample programs (provided on disk as well) are used to illustrate the concepts. For the most part, information here applies to all modes of operation.

The basic concepts of multi user access are discussed as well.

Unless you are going to restrict yourself solely to embedded SQL functions you must have a good understanding of the information provided in this section to be able to use the full functionality of c-tree Plus in any operational mode.

2. Using c-tree Plus™

This chapter discusses how to integrate c-tree Plus with your application program. An overview of the installation process is given here as well. Details on installation can be found in the **c-tree Plus Installation Guide**.

Integrating c-tree Plus™ with your program

To understand how to integrate c-tree Plus with your program, let us divide your application into three parts:

- your application code;
- the interface between your code and c-tree Plus;
- the c-tree Plus routines themselves.

Your program will make calls to the c-tree Plus functions as described in this Guide. In both Server and Non-Server single user mode, as well as Non-Server multi user mode, you will link the c-tree Plus routines with your application. In the Server multi user mode the actual c-tree Plus routines exist in a separate program, the FairCom Server. You will link a c-tree Plus interface with your application, which will take the information from your c-tree Plus function calls and transmit them to the file server.

Add #include Files

For all modes of c-tree Plus you will add a set of *#include* statements to those application modules that will be using the c-tree Plus functions. These *#include* statements provide you with access to a number of variables that are used in the c-tree Plus interface, as well as a number of useful *typedefs* and *defines*.

```
#include "ctstdr.h"
#include "ctoptn.h"
#include "ctaerr.h"
#include "ctdecl.h"
#include "cterrc.h"
```

If you are going to use the high level Incremental ISAM routines, you will also want to add the following:

```
#include "ctifil.h"
```

Use the c-tree Plus™ Functions

c-tree Plus provides a large number of functions for initializing the c-tree system, creating and opening data files and indices, and manipulating those data files and indices. You will use these functions in the appropriate places in your application modules. Within a given version of c-tree Plus, these functions are identical across all modes (single user, multi user, and server), as well as across all operating system environments (DOS, UNIX, etc.). This allows you to write one program that is portable to all of your target operating system environments without having to change the source code (other than some changes in some of the *#include* files).

Link with c-tree Plus™ Libraries

When you link your application you will include one or more c-tree Plus libraries in the process. The c-tree Plus application interface, the functions that you are calling, will always be included. The actual c-tree Plus file access routines themselves may be included or not, depending on the c-tree Plus mode that you are using. With multi user Server mode these routines are in a separate program apart from your application. In all other modes ou will include the actual c-tree Plus file access routines in your executable program.

If you use the c-tree Plus functions properly, you can change the operating mode of your application simply by linking it with the appropriate library. If you write the application so that it works in a file server mode then it can work in a multi user mode with no changes other than the library used. In addition, that same application can run in a single user mode by linking with the proper library.

Please note, however, that a number of high level functions are only available if you are using the FairCom Server. These functions are ignored by non-Server libraries. If you include multi-user parameters and functions that relate to record and file locking, these calls and parameters will be ignored by single user libraries without any adverse affect on your program.

For many of the versions of c-tree Plus the installation process will install the appropriate library file for you to use. If you are working in an operating environment in which c-tree Plus does **not** install a library, you can create your own. Information about creating these libraries can be found in the **c-tree Plus Installation Guide**.

Example Programs

After you have performed the installation and compilations as described in the Installation Guide, you should run the sample programs.

Chapter 7describes these sample programs. The first set is a simple series of programs designed to introduce you to the basic concepts of managing files and

indices with c-tree Plus. The second set contains some more complex examples which will be referred to throughout this Guide.

3. c-tree Concepts

For you to be able to understand how to use **c-tree** to it's fullest advantage, there are a number of important concepts that you need to understand. Some of these are common approaches to the C language, some of them are specific to our approach to file access using c-tree. Even if you are an experienced C programmer you should review this chapter before proceeding with c-tree.

Information about multi user access and the FairCom Server can be found in later chapters.

"C" Concepts

We assume that you are familiar with the C language. If not, you need to develop a reasonable degree of proficiency with the language before you attempt to use c-tree.

This section of the reference guide describes a number of conventions that we use in relation to the C language, as well as a few C concepts that we believe are worth reviewing.

Data Type Conventions

The actual representation of a data type declaration may vary from one operating environment or processor type to another. For instance, on one system a data type of *int* may be represented by a two byte integer, while on another system *int* may be represented by a four byte integer. Our goal is to make a product using c-tree to be as portable as possible, and differences between systems like these can circumvent that goal. We would like to ensure that what we want to be a two byte integer in c-tree will always be a two byte integer on any system c-tree is run on. Therefore, we define our own data types as shown below. By careful use of these data types you can guarantee that a program will be consistent across all operating environments that you use.

The following data types are used throughout the c-tree routines called by your application programs. C language typedefs have been included in the c-tree header files. Most of these will be found in the header file **ctport.h**.

We recommend that you use these conventions in your own products, as well. By carefully avoiding the use of native variable types in your variable declarations you can create a program that is very easy to port from one compiler or operating environment to another.

COUNT COUNT parameters are used to pass values such as key length or file number to the c-tree functions. Most c-tree functions return error codes of type COUNT. This is a two-byte signed integer.

UCOUNT UCOUNT parameters are used to pass fixed length data record lengths and file extension sizes to the c-tree functions. This is a two-byte unsigned integer.

LONG LONG values are used to represent data record positions. This is a four byte integer.

TEXT Character pointers to type TEXT are used to pass file names and key values to the c-tree routines.

IFIL Incremental ISAM structure. IFIL structures contain the parameters necessary to allow ISAM files to be created, opened and closed without a parameter file. The ISAM parameters are discussed in Chapter 5..

VRLEN VRLEN is used to pass or receive a record length and buffer sizes.

UINT UINT are native system unsigned integers.

VOID VOID is used in two ways, to define functions that return no value, and to define a generic pointer.

Some compilers do not support the declaration **void**, and others do. In *ctport.h* you will find a define similar to the following:

```
#define VOID void
```

If your compiler does not support the **void** declaration, substitute the following;

```
#define VOID int
```

VOID pointers are better than character or TEXT pointers when you are passing a variable address to a function, and you don't know if you are going to use a character string or a structure. A VOID pointer is a generic pointer, and if you are using function prototyping it will let you use any type of variable pointer. A TEXT pointer will give you a warning when you compile a program passing a structure address.

Strings

One of the shortcomings of the C language is that there is no formal data type for character strings. All string manipulation is done by processing arrays of characters. This in turn brings us to another shortcoming of the C language - there is no automatic checking on the boundaries of an array. These features can be exploited in many cases, but when it comes to string manipulation they combine to create a major problem.

When you are copying a character string from one variable to another, it is up to the programmer to ensure that the receiving variable has enough room to hold the entire string being copied. If you are not careful it is possible for the string being

copied to overlay an adjacent variable in addition to the target, which can create unpredictable results. This may be the most common error that can occur in a C program, and it can be very difficult to trace.

To complicate matters, many string manipulation operations performed in a c-tree program are NOT working with traditional null-terminated character strings. As we will see later, some types of c-tree key values may have null characters embedded in the middle. In addition, copying data structures from one location to another is handled as a string manipulation, and the data structure will have a number of null characters embedded in it.

To assist you in working with strings and structures, c-tree includes a copy function **cpybuf**. c-tree Plus redefines **cpybuf** to **memcpy** in those environments which support this ASCII function. If not, **cpybuf** is defined as:

```
VOID cpybuf(destination, source, length);
     TEXT *destination;
     TEXT *source;
     UINT length;
```

This function will copy *length* bytes from *destination* to *source*. You can copy strings, structures, or buffers with this. It will NOT stop when it sees a null character in *source*.

Another benefit of **cpybuf** is that it will stop when it has copied *length* bytes. We strongly suggest using this function when you know the length of the buffer or string that you are copying.

c-tree also provides **cpybig**, which provides the same functionality as cpybuf, but works for very large buffers even on machines with segmented memory models. The *length* parameter is declared as VRLEN instead of UINT, which provides for larger buffers.

Prototyping

If your compiler supports function prototyping we strongly suggest that you use this feature. A function declaration, or function prototype, establishes the name and return type of a function, as well as the types and number of arguments to the function. This information enables the compiler to check the types of the actual arguments when the function is used. If you accidentally use too few parameters when you invoke a prototyped function your compiler should give you a warning.

With some compilers it is very important that you use prototyping to avoid serious memory overwrites. For instance, with the Large memory model with the Microsoft C compiler, and functions that use TEXT pointers. If you do not include a function prototype in your code, the compiler will have to determine if it should use a 16 bit or 32 bit address for the function arguments. If it chooses the wrong one the function may copy the information to the wrong location in memory. By including the

function prototype in your code you inform the compiler of the type of parameter that will be used, and the correct address will be passed.

The header file **ctdecl.h** contains declarations for the commonly used c-tree functions. There are two sets of declarations, one with prototypes and one without. In the header file **ctoptn.h** as it is shipped the value PROTOTYPE is defined. This will include the prototype definitions. If your compiler does not support prototyping then you should remove the PROTOTYPE define from **ctoptn.h**.

c-tree Plus™ Keys

Key values permit data records to be located quickly, avoiding lengthy sequential searches. c-tree maintains keys in an **index** file which can be rapidly searched. Each key in c-tree index file also has a data record pointer associated with it.

Target key values are passed to the c-tree routines by a pointer to the first byte of the key value. Keys in c-tree should **not** be handled as normal character strings. No restrictions are placed on the individual byte values comprising the key. In particular, a null or zero byte *DOES NOT* signify the end of a key value. Therefore, you should use a function like **cpybuf** to manipulate keys.

When you define a key value to c-tree, you will establish the length of the key. c-tree does not pad the key values to their full length for you. It assumes that you have completely filled in the key value to its defined length (specified at the time the index file is created). If you are not careful to pad your keys to their full length you may find that your index searches will not work the way you expect.

Alternative Key Types

c-tree supports a number of basic key types:

- fixed length keys (type 0);
- leading character compression (type 4);
- padding compression (type 8);
- leading/padding compression (type 12, a combination of types 4 and 8).

Prior versions of c-tree allowed additional key types. Refer to the document *Differences from Prior Versions* for information on these types.

Normally you will use fixed length keys. If there is a reasonable amount of duplication in the leading characters of the key values, you can decrease the size of your index file by choosing leading character compression. If your keys vary in size you may find that padding compression will decrease your index size (remember that you should be padding keys to the same length). Key type 12 combines both types of compression.

Using one or both of the compression modes increases the CPU time to process a key since the key values must compressed and expanded, and the key values cannot be located at fixed positions within the node. The increased CPU time is eventually offset by the reduced disk head movement due to the reduced size of the index. As the index grows, the compression can lead to a reduction in the number of levels in the tree structure (compared with a fixed length tree) because of the increase in branching (due to more keys in each node). A reduction in the number of levels leads directly to a reduction in the number of disk accesses.

Leading character compression and padding compression each require a byte to store the extent of the compression. Hence, if either or both of the compression modes are in use, the maximum key length is restricted to 255. Further, employing the compression modes when no compression is possible will cause wasted storage since the bytes used to note the extent of the compression will be stored in addition to the key values.

The default padding character is an ASCII blank (20 hex). This can be changed for a particular index file by using the **SetVariableBytes** function (see the description of this function in the c-tree Plus Function Reference).

Fixed Length Keys

This is the simplest key type. If you are not sure what key type to use, choose this. Key type 0 is for fixed length keys that have little chance for significant leading character redundancy. Even if there is a possibility of key compression, you may still choose key type 0 if you wish to avoid the increased code size (about 4k bytes) or increased computation required to process the compressed keys.

Leading Character Compression Keys

Key type 4 is for fixed length keys which are likely to have significant leading character duplication among the key values. The common beginnings of successive key values are represented by a compression byte count. Consider the following sequence of key values:

```
"AABBCCDD"
"AABBCDEF"
"AABCDEFG"
```

With leading character compression enabled, these 24 bytes could be represented by the following 19 bytes:

```
"AABBCCDD"
4"CDEF"
3"CDEFG"
```

The integers preceding the key values indicate the number of leading characters which the key value shares in common with the immediately preceding key value.

Padding Compression Keys

Key type 8 is for variable length keys for which not much leading character duplication is expected. Trailing padding compression enables variable length keys to be stored in the index without incurring the wasted storage of fixed length keys (with the key length set to the maximum possible length). Assuming a maximum key length of 15 bytes, then the fixed length representation of the following key values

```
"ABCDEF         "
"DEFGHIJK       "
```

requires 30 bytes, while the compressed form would save 14 bytes:

```
"ABCDEF"9
"DEFGHIJK"7
```

Combined Compression Keys

Key type 12 provides the maximum key compression; combining both leading character and padding compression.

ISAM Keys

If you are using the c-tree ISAM routines, the key value definitions are provided in a parameter file or an incremental ISAM structure. The keys can be compound keys formed from various fields in the data record. Individual segments of the key are defined by the key segment parameters, as discussed on page 5-14.

Duplicate Keys

When you create a new index you can set the index up for keys that are always unique, or you can allow duplicate keys. If the index file is set up for duplicate keys, then the four byte data record position is automatically appended on the right end of the key to create a unique key value. The programmer must remember to increase the key length by four bytes to allow for this addition.

This suffix *DOES NOT* appear in your data record. You *DO NOT* create extra fields in your record structures to accommodate the suffix.

For example, a key length of twelve will have space for eight bytes of an actual key value and four bytes for the automatic record position suffix.

When a key value is returned from one of the low level index file search routines (e.g., **NextKey**), the last four bytes of the returned value will contain the associated data record position. Recall that the key value found in the index by a search routine is "returned" by placing a copy of the entry into the area pointed to by the idxval parameter. When using a value found in an index that supports duplicate keys, be sure to strip off the last four bytes before displaying the key (or using it for a computation if it is a numerical key) since the suffix is not a proper part of the key value.

Sequence Numbers

c-tree supports another mechanism to distinguish among key values: the ISAM level segment mode value of 3 (see Chapter 5.) indicates that the key segment will automatically be filled-in with a four byte ever increasing sequence number. Each time a record is added to the file, the sequence number is incremented. Therefore, instead of specifying that the index supports duplicates, the index should be set for unique keys, and a four byte segment defined as part of the key with a segment mode of 3.

NOTE: unlike the duplicate key approach above, this approach *DOES* require that a four byte region of the file be reserved for the sequence number which c-tree will automatically initialize.

Data Records

Data record positions are absolute byte offsets from the beginning of the data file (where the first byte of the file is at offset zero). For fixed record length data files, the byte offsets corresponding to the beginning of each record are guaranteed to be a multiple of the record length specified for the data file at the time of creation. Therefore, a data record position in a fixed record length data file can be converted to a relative record number by dividing the data record position by the record length.

Buffer Regions

As a reminder, many c-tree functions require you to pass a pointer to a "buffer" area where either a data record or key value will be written. c-tree has no way to determine if the area in which it will write the requested information is actually large enough to contain the information. If it is not, c-tree will contaminate data and/or executable code. *CAREFULLY* review all areas used as buffers to ensure that they are sufficiently large.

Data Record Delete Flag

c-tree uses the first byte of **fixed** data records for a delete flag. Whenever a fixed length data record is deleted (either as a result of a call to **DeleteRecord** or **ReleaseData**) the first byte is set to 0xff. The rebuild and compact utilities check

this first byte to determine if a record is active or deleted. Also, whenever a record is about to be reused by **NewData** (which is automatically called by **AddRecord**), the first byte is checked to be sure that an active record is not about to be reused by mistake.

It is not necessary to reserve this byte if your application stores only ASCII characters in the first byte; however, you shouldn't store binary data in the first byte.

Resource Records in Fixed Length Files

Resource records (see chapter 8) that are embedded in files with **fixed** data records have a first byte of 0xfe.

Data and Index Files

File Modes

When c-tree files are created and/or opened, their characteristics are partly determined by the *filmod* parameter. *filmod* determines:

- if a file is opened in a standard manner or in a special manner which avoids problems with limited file descriptors;
- if a file is to be shared among multiple users or used exclusively by one user; and
- if a data file is comprised of fixed length or variable length records.
- the method of transaction management for the file.
- if a file is to contain Resources.
- if a file is to be a Superfile.

Some of these modes will be discussed in more detail in later chapters.

Virtual File Open

PERMANENT
VIRTUAL

Many operating systems and/or C compiler run-time libraries limit the number of files which can be opened at one time. In order to free the application code from worries over the number of files opened at one time, c-tree implements two types of file opens: PERMANENT and VIRTUAL. A permanent file open causes the file to be opened and stay open until the program executes a file close. This is the traditional type of file open. A virtual file open causes the file to be opened, but allows it to be transparently closed and reopened in order to allow other virtual files to be used.

When it is necessary for a virtual file to be temporarily closed, c-tree selects the least recently used virtual file. This file will remain closed until it is next used, at

which time it will be automatically reopened. This strategy causes c-tree to use all available file descriptors.

This activity is transparent to both the user and the programmer. You do not have to treat virtual files any differently than you would a permanent file. There are only two effects from using virtual files: the user may notice a slowdown in performance when the virtual files are being opened and closed,; the programmer will be less likely to have to deal with run-time errors caused when more files are opened than the operating environment supports.

For example, in MS-DOS there is a limit of 20 file handles available to your application, if you have set the FILES = statement to 20 or larger in the CONFIG.SYS. file. Some of these file handles are already consumed by the standard input and output. If you have an application that needs more files open you will run into this limit. By setting the *filmod* of a number of your files to VIRTUAL, c-tree will allow you to have more files open than the limit.

Fixed versus Variable Length Records

FIXED
VLENGTH

When a data file is created, the *filmod* parameter is used to specify whether the data records are fixed or variable length. Once a data file is created, its record length characteristics cannot be changed. Since variable length records require a little more processing and some extra code, it is advisable to use fixed length records unless variable length records are actually necessary.

The file modes are FIXED and VLENGTH, which can be used with the other file mode constants.

Fixed length data files can be created with record lengths in the range from 5 bytes to 65,535 bytes.

Variable length data files are created with a defined minimum record length. The minimum record length specifies the smallest data record which will be added to the file. Conceptually, this minimum record length corresponds to the fixed length fields which may be part of your record structure. The minimum fixed-length portion may be set to zero. The maximum fixed-length portion is 65522. The total length of a variable length records is unlimited (as long as you have enough buffer space in your application to hold the record).

If you are not going to use a field in the variable length portion of the record to build an ISAM key you can put any type of field there that you wish. If, however, you are going to build an ISAM key with one of these fields you have to be careful. The simples approach is to have all of the fields in the variable length portion of the record be themselves variable length fields, terminated with a delimiter. This delimiter typically is a NULL character, but that can be changed by the **SetVariableBytes** function. You can, however, place any type of filed in the variable length portion of the record if you use a **Record Schema**, as explained on page D-1.

Multi-User File Mode

EXCLUSIVE
SHARED
READFIL
CHECKLOCK
CHECKREAD

When c-tree is used in a network or multi-tasking environment it is necessary to specify whether individual files are shared among the users or opened exclusively for one user, and whether or not the FairCom Server should check for proper lock ownership.

How to use these modes is explained in detail on page 6-1 and page 10-2.

Transaction mode

PREIMG
TRNLOG

With the FairCom Server you can specify the method of transaction management for this file. Transaction management is discussed in more detail in Chapter 11.. Details on transaction modes can be found on page 10-2.

I/O Management

DUPCHANEL

With the FairCom Server you can control certain how the server uses I/O channels. This mode is discussed on page 10-3.

WRITETHRU

WRITETHRU forces the operating system to flush all disk cache buffers when a data write occurs. This parameter can slow performance of the file handler down. On the other hand, it is an important feature to use if you want to ensure that all data writes are put to the disk immediately. It is particularly important if you are in an operating environment where the system crashes a lot, and you are not using transactions and the TRNLOG mode. However, WRITETHRU does not guarantee that operating system buffers have been flushed.

If automatic recovery is active for a file you do not use this mode.

Resources

DISABLERES

Resources are special variable length records that can be used to store information in a data file that is not in the same format as the regular data records. Normally they are enabled, but under certain circumstances you may want to disable them. See Chapter 8

Superfiles

SUPERFILE

A Superfile is a c-tree file that may contain an arbitrary number of c-tree data and index files. This permits a group of files to be moved or copied as a single physical unit. See Chapter 8.

filmod Values

At file creation and open, the *filmod* parameter is used to specify the important file characteristics discussed above. Due to the number of different options it is not practical to show you all possible combinations.

When you are asked to supply the *filmod* parameter you can OR the various options together. For example;

```
(VIRTUAL | EXCLUSIVE | FIXED)
(PERMANENT | SHARED | FIXED | TRNLOG | DUPCHANEL | WRITETHRU)
```

Certain options are mutually exclusive, so they should not be used together. Only one option should be used from each of the following groups;

```
VIRTUAL or PERMANENT
EXCLUSIVE, SHARED or READFIL
FIXED or VLENGTH
PREIMG or TRNLOG
```

When creating a data or index file you MUST specify one option from each of the FIRST THREE groups shown above.

Data and Index File Numbering

Once a file is opened for use it is referred to by its file number. Therefore, each data and index file must be assigned a unique number by the programmer. Data files, index files and index members all draw file numbers from the same pool; therefore, data files and indices cannot have the same number assigned. An index member is an index which is stored in the same physical file with other indices.

The first additional index member in an index file is given a *keyno* one greater than the *keyno* of the host index. The second index member in an index file is given a *keyno* two greater than the host **keyno**, and so on.

NOTE: multi-user, non-server implementations which require a dummy lock file may have additional file numbering constraints. See Chapter 6..

Low Level Functions

c-tree Plus provides you with a full range of low level functions. These functions provide you with complete access to all of the features of c-tree Plus. You may require specific control over the data and index file manipulations, managing each data and index file individually.

ISAM functions

c-tree provides a set of high level functions for manipulating your data and index files that are called the ISAM functions. These functions perform multiple file and index access operations for each function call.

The ISAM routines assume that the relationship among the data and index files is hierarchical in the sense that each data file may have many indices, while each index relates to only one data file.

For example, an accounts payable system may define a vendor data file indexed by vendor name and number, and an invoice data file indexed by vendor number and invoice number. In this case each of the data files has two indices. Even though each of the data files is indexed by vendor number, there is a separate vendor number index for each data file.

The ISAM routines also assume that the key values for each index can be derived from the contents of the data records of the associated data file. As described later, the Key Segment parameters specify how to extract a key value from a data record.

To give you an example of how the ISAM functions perform multiple actions, consider the following sequence of events. We want to open a data file and its two associated indexes, find the first key value in that index, and read the associated data record into memory. With the low level c-tree functions the steps would be (the c-tree function used is in parentheses):

- Initialize c-tree (**InitCTree**);
- Open the data file (**OpenFile**);
- Open the first index file (**OpenFile**);
- Open the second index file (**OpenFile**);
- Find the first key (**FirstKey**);
- Read the record (**ReadData**).

The same procedure, using the ISAM functions, would be as follows:

- Initialize c-tree, open the data file and the two indices (**OpenISAM**);
- Find the first key and read the record (**FirstRecord**).

As you can see, the ISAM functions can save you work, and make your code easier to follow.

Error Handling

c-tree functions generally return either an error code or a data record position. Functions which return an error code return a value of zero to indicate no error has occurred, or a non-zero value which designates the type of error. Error codes are found in *cterrc.h*. Functions which return a data record position return a zero value (which is never a legitimate data record position) to indicate an error or unsuccessful index search.

uerr_cod For low level functions, the actual error code can be found in the global variable **uerr_cod.**

isam_err ISAM functions place the actual error code in the global variable **isam_err.** Since **isam_fil** ISAM functions work with multiple files you can check the global variable **isam_fil** for the file number that the error occurred with.

File Recovery

Header Record

Information about each data file and index is stored in the first record of the file, called the **header record**. This information is modified whenever you add a record or key. In many cases this information is not written back to the header record on disk immediately, particularly in single user situations. This improves the speed of the c-tree system, but also makes it possible to have incorrect information in the header if the program crashes before the header is updated.

If this situation occurs, c-tree is able to detect that the header has been compromised. An appropriate error code will be returned to you if you try to open a compromised data file or index.

c-tree provides a high level function **RebuildIFile**, which will rebuild the data file header (and other information) as well as recreate the index. To use this function you must build an incremental ISAM structure for the file and indices, even if you are not using ISAM functions.

If your keys cannot be described properly in an incremental ISAM structure you will have to rebuild the index yourself. A general process would be:

- Use **RebuildIFile** to rebuild the data file;
- erase the damaged index;
- sequentially process the data file, adding the keys to a new index.

4. Using Low Level Functions

To introduce you to how to use the c-tree Plus low level functions we are going to build a very simple application. A listing of this application can be found on your distribution disk in the file *lowlevel.c*. It is also explained in more detail in Chapter 7. In this chapter we will present an outline of the program. Following the application we will review the other low level functions that are available.

Full explanations of each function can be found in the **c-tree Plus Function Reference Guide**, along with brief examples of how to use them.

A Simple Application

We will start with a simple file maintenance application with a single data file and a single index file. The records are fixed length, and we will not allow duplicate keys.

Application Outline

An outline of our application would be:

- initialize the file system;
- ask if the user wants to Add, Delete, or Modify a record;
- when done, close the file system.

The Application

At the top of each application you should have the following include statements:

```
#include "ctstdr.h"
#include "ctoptn.h"
#include "ctaerr.h"
#include "cterrc.h"
#include "ctdecl.h"
```

Initialize c-tree Plus™

InitCTree

The first step needed is to initialize the c-tree system. This is done with a call to the **InitCTree** function. Amongst other operations, this will allocate memory for buffers and files. This function requires three parameters; the number of index buffers, the number of data and index files, and the number of node sectors.

The number of buffers to enter depends on several factors. First, if you are running a Non-server multi-user system, you should only enter 6. More will not help, unless

you have important read-only indices. For single user or server systems, reasonable speed is achieved with 6 buffers for each active index.

The second parameter is the number of data files and indices that you will have open at any given time.

The most common value for the third parameter is **four**. For further information on this parameter see the listing for **InitCTree** in the **c-tree Plus Function Reference Guide**.

OpenFile
CreateDataFile
CreateIndexFile

Once c-tree has been initialized, you must open the data files and indices. **OpenFile** is used to open an already existing data and index file (one call for each data file and each index). If the data file or index does not exist, you must use either **CreateDataFile** (for data files) or **CreateIndexFile** (for index files) to create them.

Add a record

If the user wants to add a record, we must first see if the key already exists (unless you want to allow duplicate keys), then we need to add the key and write the record.

GetKey

First we look to see if the key exists by using the **GetKey** function. This will return the record number of the key if it is in the index file, so if a record number is returned we must tell the user that they cannot add it again.

NewData

To add a record to a data file we must ask c-tree Plus for the next available data record. This can be a record added to the end of the file (c-tree will automatically extend the file as necessary). It can also be a record that had been used before, but was deleted. c-tree keeps track of records that have been returned to the data file, so that when you call **NewData** to get a record c-tree will first look to see if there is a deleted record to give to you before it will extend the file further.

WriteData

Once we have a data record number we can actually place the data in the file with **WriteData**.

AddKey

This function is used to add the key to the index file. We do this **after** we write the record for two reasons. First, **AddKey** must know what the record number is that is associated with this key. Second, we want to be sure that the data record can actually be written before we put the key into the index.

It is important that the key be padded to it's full length, so that it can be easily found later. For instance, if the key length is set to 25 characters, but you give **AddKey** a null terminated string that is shorter than 25 characters, you may have a problem later. The full 25 characters will be added to the index, including the undefined characters after the null. Since you don't know what the characters after the null are, you will not be able to make an exact match later.

Find a record

To find a record we must look for the key in the index and then read the associated data record. In this simple example we will look for an exact match of the key that the user enters. More sophisticated searches can be done for partial keys, or scanning up and down through the index.

GetKey This function will look for an exact match of the key, and return the record number if the key is found. Remember that the key must be **perfectly formed** for an exact match. This usually means that it must be padded the same way that the key was when it was entered into the index.

ReadData Once the key has been found you use **ReadData** to read the associated data record.

Delete a record

DeleteKeyBlind
DeleteKey There are two functions that can be used to delete a key, **DeleteKeyBlind** and **DeleteKey**. All that **DeleteKeyBlind** requires is the key to delete, while **DeleteKey** also asks for the associated record number. If you are allowing duplicate keys you must use **DeleteKey**.

ReleaseData Once the key has been deleted from the index, you will want to delete the data record as well. **ReleaseData** marks a data record as deleted. Now this record is available to be reused by a subsequent call to **NewData**.

Close the system

When you are ready to exit your application you will want to close the data files and indices. If you don't, and you have added or deleted records or keys, you will have damaged your files. Closing the files is necessary to update the header information to the file.

CloseFile This function is called for every open data file and index.

Low Level Function Overview

Some of the low level functions (those which return type COUNT) return an error code value. A zero error code indicates success. The rest of the index file functions (those which return type POINTER) return a data record position. A zero data record position indicates an error. Each low level function sets the value of a global error code variable: uerr_cod.

Several of the functions listed here have two versions - a standard version and an extended version. The extended version has the same name as the standard version, with a suffix of **Xtd**. With the extended version you are able to access the security features of the FairCom Server.

System Functions

InitCTree
InitCTreeXtd
This function must be called before using any c-tree functions (unless you are using ISAM functions). It initializes the c-tree buffers for the number of files that you are going to use. If you are using the FairCom Server, it also logs you on to the server.

StopUser
If you are using the FairCom Server, and you are not using the ISAM functions, you must call this function when you are finished using the Server, typically just before you exit the application. If you are not using a Server this function is ignored.

File Manipulation Functions

CloseFile
Use this to close a data or index file when you are finished with it. It is important to close any file if it has been updated. If you do not close an updated file when your application terminates, the file will be damaged.

CreateDataFile
CreateDataFile-Xtd
This function is used to create a data file. The characteristics of the data file are set at this time, such as the record length and file mode (fixed or variable length). Note that c-tree does not require that you establish the length of the file itself. As you add records to the file the length is automatically extended.

CreateIndexFile
Index files are created with **CreateIndexFile**. The characteristics of the index file are set at this time, such as the key length.

CreateIndex-FileXtd
CreateIndex-Member
Any physical index file can contain multiple logical indices. While there is only one file on the disk, it can look to your program as if it is several separate indices. This is handy if you have a limit to the number of files that can actually be open. **CreateIndexMember** is used to establish multiple indices within one index file.

DeleteFile
This function is used to delete a specified file. It is better to use this function than a C language function such as **unlink**, since in Server environments the application may not have direct access to the file.

GetFileInfo
This function returns information from the data file or index header record.

GetSymbolic-Names
GetSymbolicNames returns symbolic information from the file, such as the file name.

OpenFile
OpenFileXtd
Use this function to open a data or index file that has already been created with **CreateDataFile** or **CreateIndexFile**.

UpdateFile-Mode
This function is used to change critical file mode attributes.

Security
Use this function to alter a password, Group ID, or permission mask of a file.

Data Record Manipulation

NbrOfRecords Get the number of active records in a fixed length data file.

VRecordLength This function returns the length of a variable length data record.

LockData **LockData** is used to lock and unlock data records.

NewData This function will return the next available record number in a **fixed** length data file.

NewVData This function will return the next available record number in a **variable** length data file.

ReadVData Use this function to read a variable length data record. You should use **VRecord-Length** before calling **ReadVData** to ensure that your buffer is large enough to hold the record.

ReadData Use this function to read a fixed length data record.

ReleaseData **ReleaseData** deletes a fixed length record from a data file, marking it as available for use for the next call to **NewData**.

ReleaseVData This is similar to **ReleaseData**, except that it is used to delete a variable length data record.

WriteData Writes a fixed length data record to the file.

WriteVData Writes a variable length data record to the file.

Key Manipulation

AddKey Add a key value to an index.

DeleteKey Deletes a key value from an index. You must pass **DeleteKey** the proper data record number for this to work successfully.

DeleteKeyBlind Deletes a key value from an index. This does not require the data record number.

GetKey Search the index for an exact match of the given key. Do not use this in indices that allow duplicate keys.

EstimateKey-Span This finds the approximate number of entries between two key values

KeyAtPercentile Finds the key value located approximately at the percentile specified.

FirstKey Find the **first** key in the specified index.

GetGTEKey Find a key that is **equal to or greater than** the target key.

GetGTKey Find the first key that **follows** the target key.

NbrOfKey-Entries Get the number of entries in an index file.

LoadKey Loadkey is used to build an index with multiple key values when the key values are already in sorted order. It works faster than using **AddKey**, but has some restrictions.

LastKey Find the **last** key in the index.

GetLTEKey Find a key that **precedes or equals** the target key.

GetLTKey Find a key that **precedes** the target key.

NextKey Find the next key in the index, relative to the last key search. Do not use this if you are updating the index, or if you are in a multi user environment.

PreviousKey Find the previous key in the index, relative to the last key search. Do not use this if you are updating the index, or if you are in a multi user environment.

FairCom Server Transactions

If you are using the FairCom Server there are a number of functions available to perform transaction processing. They are only available if you are using the Server. If you are in single or standard multi user mode they will be ignored. See chapter 11 for information on transaction processing.

Abort Aborts an in-progress transaction, releasing all locks.

Begin Begin a transaction. A transaction number is assigned. This must be followed by a call to **Commit** to post the transaction completely.

Commit Commit the transaction started with a **Begin**.

RestoreSave-Point Undo operations back to a specified save point, as created by **SetSavePoint**. This allows you to return safely to a certain point without having to abort the entire transaction.

SetSavePoint Establishes a save point. This provides you with a partial undo capability, by using the **RestoreSavePoint** function.

ClearTranError Clears the transaction error flag that is set if there has been an error in an update function. This allows the Commit function to complete successfully, but it should be used with extreme caution.

Resource Functions

If you are using the FairCom Server you will be able to use Resources. A Resource is auxiliary information stored in a file which is transparent to the regular processing of the file. Resources can be used to store version information, counters, or any other type of information related to an application which is not part of the regular data record contents.

AddResource Adds a Resource to a data file.

DeleteResource Deletes a Resource from a data file.

Enable-Resource Enables Resources in a file that was originally created without Resources being enabled.

GetAlternate-Sequence If you have stored an alternate collating sequence as a resource, you can retrieve the information with this function.

GetDODA Use this function to retrieve DODA and record schema information that has been stored in a file as a Resource.

GetResource Retrieves Resource information from a data file.

PutDODA This function is used to store a DODA (as explained further on page D-4) in a file as a Resource.

SetAlternate-Sequence This allows you to define an alternate collating sequence for a file, stored as a resource.

SetVariable-Bytes Change the value for the Padding character and Field Delimiter for a file. This is stored as a resource.

Update-Resource Updates Resource information in a data file.

5. ISAM functions

c-tree provides a set of high level functions for manipulating your data and index files that are called the ISAM functions. These functions perform multiple file and index access operations for each function call.

The ISAM routines assume that the relationship among the data and index files is hierarchical in the sense that each data file may have many indices, while each index relates to only one data file.

For example, an accounts payable system may define a vendor data file indexed by vendor name and number, and an invoice data file indexed by vendor number and invoice number. In this case each of the data files has two indices. Even though each of the data files is indexed by vendor number, there is a separate vendor number index for each data file.

The ISAM routines also assume that the key values for each index can be derived from the contents of the data records of the associated data file. As described later, the Key Segment parameter specifies how to extract a key value from a data record.

A Simple Application

We will use a simple file maintenance application as an example, as we did with the Low Level Functions. The records are fixed length, and we do not allow duplicate keys. The complete source code for this simple application can be found on your distribution disk in the file *isam.c*. In addition, it is explained in more detail in Chapter 7. We will just outline the application here.

Application Outline

The outline, as with the low level functions, of our application is:

- initialize the file system;
- ask if the user wants to Add, Delete, or Modify a record;
- when done, close the file system.

ISAM Parameters

The characteristics of the data files and indices used in your application program can be defined in two different ways, either with a **ISAM Parameter File**, or with **ISAM Structures** defined in your application. The ISAM Parameter File is a simple ASCII file that you can create with your text editor. It contains records that specify the characteristics of all data files, indices, and keys, for a particular application. The **OpenISAM** and **CreateISAM** functions use this file to initialize the system.

Alternately, you can specify the structure of each data file and it's associated set of indices in a set of structures in your application. Each data file (and it's associated indices) are created or opened with the **CreateIFile**, **OpenIFile**, and **OpenFile-WithResource** functions. Initially we will discuss using an ISAM Parameter File. Later we will discuss the use of structures (referred to as Incremental ISAM Structures).

The Application

At the top of each application you want to have the following include statements to be able to use c-tree:

```
#include "ctstdr.h"
#include "ctoptn.h"
#include "ctaerr.h"
#include "cterrc.h
#include "ctdecl.h"
```

If you are going to use Incremental ISAM structures you need to add the following include:

```
#include "ctifil.h"
```

Initialize c-tree Plus™.

OpenISAM

The first step needed is to initialize the c-tree system. This is done with a call to the **OpenISAM** function. This will take the information stored in the ISAM Parameter File, allocate memory for buffers and files, and open all of the data files and indices.

CreateISAM

If the data files and indices do not already exist, you will use the **CreateISAM** function to create them.

Add a record

AddRecord

In the low level function example we had to call three functions to add a record to the file system; **NewData** to get the data record, **WriteData** to write the record, and **AddKey** to add the key to the index. With the ISAM functions the process is much simpler. All that is needed is a call to **AddResource**. This performs all of the functions that are necessary to add the record and key.

Find a record

GetRecord

To find a key and read the associated record into our application we simply make a call to **GetRecord**. This function finds the match to the key and reads the record into your data buffer. More sophisticated searches can be done for partial keys, or scanning up and down through the index.

Delete a record

DeleteRecord

This function will delete a record from the data file, and it's key from the index. The record that is deleted is the "current ISAM record", which we will define later in this chapter.

Close the system

When you are ready to exit your application you will want to close the data files and indices. If you don't, and you have added or deleted records or keys, you will have damaged your files. Closing the files is necessary to update the header information to the file.

CloseISAM

This function only needs to be called once. Every data file and index that was opened with the call to **OpenISAM** will be closed with this one function call.

ISAM Error Handling

All of the ISAM functions return an error code value. Zero return indicates success, otherwise an error has occurred. Each call to an ISAM function sets the values of two global variables:

- *isam_err* - this contains the error code value;
- *isam_fil* - this contains the data or index file number involved in the error.

Since each ISAM function can work with multiple files you need to check the value of *isam_fil* to see which file caused the error.

Current ISAM Record

When using the ISAM functions, the concept of the current ISAM record for a data file is very important. c-tree remembers the record position of the last data record found for each ISAM data file, as well as information about the ISAM keys for that data record. This last record found is referred to as the "current ISAM record".

Many of the ISAM routines use the current ISAM record. For example, the **DeleteRecord** function deletes the current ISAM record, and **NextRecord** uses the current record information to help find the next record in key sequential order. If no current ISAM record exists (as happens just after the files are opened), but is required, then the ISAM function will return error code ICUR_ERR.

In older versions (4.3 and before) of c-tree you had to be very careful with record rewrites (**ReWriteRecord** and **ReWriteVRecord**). c-tree would use your buffer area to determine the ISAM key structure for the current record. If you changed

one of the fields that was used as a part of the key, and then rewrote the record, you could damage the index files.

With c-tree Plus you no longer have to be as careful with record rewrites. c-tree keeps it's own image of the key fields. In a rewrite, c-tree determines if a key field has changed by comparing the contents of it's current key image buffer with your data record buffer. If a key has been changed, the old key is deleted and the new one is added. This is automatically done for ALL indices relating to this record.

This does not mean, though, that you don't have to be concerned with ISAM record rewirtes. After a call to **ReWriteRecord** or **ReWriteVRecord**, the current ISAM record is now the **updated** record. This may cause problems when your application is moving through a file in index sequential order while making updates. If the key field is updated, then the NEW current ISAM record may be substantially behind or ahead of the prior current ISAM record before the update was made. If, after an update, you wish to continue sequentially from the original record, do the following in your application:

```
ResetRecord(datno,SWTCURI);
```

Place this function call **after** your rewrite operation, and before any other ISAM function. This restores the current ISAM record to be the original one before the rewrite.

Note that the same technique may be used with variable length records. But beware that since the record may be moved physically within the file if its size is extended during the rewrite, it is possible to encounter the record again. Therefore, it may be necessary to "mark" scanned records.

Other functions that can be used for accessing or modifying the current ISAM record in c-tree Plus are;

- **CurrentFileOffset**
- **SetRecord**

ISAM Function Overview

Several of the functions listed here have two versions - a standard version and an extended version. The extended version has the same name as the standard version, with a suffix of **Xtd**. With the extended version you are able to access the security features of the FairCom Server.

ISAM Parameter File System Functions

These functions work with the information found in the ISAM Parameter File.

CloseISAM — This function will close all of the data files and indices that were opened with a call to **OpenISAM**, **CreateISAM**, **OpenIFile**, and **CreateIFile**. One call to the function is all that is necessary. If you are using the FairCom Server this call will also log you off of the Server.

CreateISAM
CreateISAMXtd — This function will create all data files and indices described in the ISAM parameter file. If you are using the FairCom Server this call will also log you on to the Server.

OpenISAM
OpenISAMXtd — Use this function to open all data files and indices described in the ISAM parameter file. The files and indices must have already been created. If you are using the FairCom Server this call will also log you on to the Server.

Incremental ISAM System Functions

These functions work with Incremental ISAM structures that you establish in your application, rather than with an ISAM parameter file. (See also in the previous section; **CloseISAM**)

CloseIFile — Closes a particular data file and it's associated indices, based on an incremental ISAM structure.

CloseRFile — Closes a data file and it's associated indices that have been opened with the **OpenFileWithResource** function.

CreateIFile
CreateIFileXTD — Creates a data file and it's associated indices, based on an incremental ISAM structure.

GetIFile — Retrieve the IFIL structure for a file from the file's resource.

InitISAM
InitISAMXtd — Prior to calling **OpenFileWithResource, OpenIFile** or **CreateIFile** you must initialize the c-tree system. This will allocated index buffer and file handle space for the files opened as a part of the incremental ISAM system. If you are using the FairCom Server this call will also log you on to the Server.

OpenIFile
OpenIFileXTD — This function will open an existing data file and it's associated indices based on an incremental ISAM structure.

OpenFileWith-Resource
OpenFileWith-ResourceXtd — This is similar to **OpenIFile**, but instead of using an incremental ISAM structure declared in your program it will use similar information that has been stored as a Resource in the data file when it was created with **CreateIFile**.

RebuildIFile
RebuildIFileXtd — If a data file or index becomes corrupted, this function can be used to recover them. Based on the information found in an incremental ISAM structure, the function will go through every record in the data file, recreating all required keys, and will then rebuild the data file header record.

Data File and Index Functions

AddRecord	This adds a fixed length data record to a file, as well as updating all associated indices.
AddVRecord	This adds a variable length data record to a file, as well as updating all associated indices.
DeleteRecord	Deletes the current fixed length data record, and removes the appropriate keys from the indices.
DeleteVRecord	Deletes the current variable length data record, and removes the appropriate keys from the indices.
GetRecord	Looks for the target key in an index, and reads the matching data record.
FirstRecord	This can read either the first physical record in a data file, or the record that has the first key in a particular index.
CurrentISAM-Key	Supplemental function for building an ISAM key from a data record image.
CurrentFile-Offset	Retrieves the byte position of the current ISAM record.
VRecordLength	Returns the length of current ISAM variable length data record.
GetGTERecord	Reads the record with a key that is greater than or equal to the target key.
LockISAM	Enables or frees locks.
LastRecord	This can either read the last physical record in a data file, or the record that has the last key in a particular index.
GetLTERecord	Reads the record with a key that is less than or equal to the target key.
NextRecord	Reads the next record in the data file, either in sequential physical order or key order. This is relative to the current ISAM record.
PreviousRecord	Reads the previous record in the data file, either in sequential physical order or key order. This is relative to the current ISAM record.
ReReadV-Record	Functions that read the record, such as **FirstRecord** or **NextRecord**, read only the fixed length portion of the data record. **ReReadVRecord** is used to read the entire variable length record. Make sure that your buffer is large enough by getting the record length with **VRecordLength**.

ReReadRecord Reread current ISAM record. Will only read the fixed portion of a variable length record.

ReWriteRecord Rewrite the current fixed length data record.

ReWriteV-Record Rewrite the current variable length data record.

SetRecord Allows you to alter the current ISAM record number and data record image location.

TransformKey The ISAM parameters for your application specify how a key is to be built. This may include translations of segments of the key. You can use **TransformKey** to build a key according to the ISAM parameters, if you need a key target value for some function.

ResetRecord Updates the current ISAM record. Use this to manipulate the c-tree internal representation of the current ISAM record, such as restoring the current ISAM record to the record position before a rewrite.

Set Functions.

A Set is a group of records that have keys that match a certain target. These functions are used to define and access Sets.

For example, an index contains the following keys:

```
CASE
DISKCASE
DISKDRIVE
DISKETTE
KEY
KEYBOARD
KEYCAP
```

You can use functions like **FirstRecord** and **NextRecord** to scan through the index, looking for records. These functions work on the entire index file. **FirstRecord** would find the first key in the file, "CASE", and **NextRecord** would give you each of the following keys in turn, on through "KEYCAP". You could also define a Set, with a target of "DISK" and a target length of 4. The function **FirstInSet** would find the key "DISKCASE" as the first record, and **NextInSet** would give you each of the following keys in turn, with "DISKETTE" being the last record it would find. This allows you to establish subsets of an index, and process them as if they were a separate index in themselves. Normally one set is worked with at a time, as established by the function **FirstInSet**. However, it is also possible to define and manage multiple sets simultaneously. To do this you will have to allocate space for

the multiple sets with **AllocateSet**, and before each Set function you will have to specify which Set you are working with by using the **ChangeSet** function.

AllocateSet Use this function to allocate key buffers for multiple, simultaneous Sets. If you are going to use one Set at a time you do not need to use this function.

DoBatch This function is used to perform operations on a Set of records. For instance, to delete all records of a Set, to read all records of a Set into a buffer, or to count the number of records in a Set.

ChangeSet If you are using multiple, simultaneous Sets, you must use this function to specify which set is going to be used in a Set function call.

FreeSet This function frees the memory allocated by **AllocateSet**. If you need to change the number of Sets you have room allocated for, you must first call **FreeSet** to free the existing memory, then **AllocateSet** to establish the new Set buffers. **CloseISAM** will also free the Set buffers.

FirstInSet This establishes a Set, based on a target key value and length. It also reads the first record of the Set into your buffer, making it the current ISAM record.

LastInSet This will read the last record in a Set.

PositionSet This function establishes a Set, as does **FirstInSet**, but the Set is based on the key value for the current ISAM record, rather than on a target value passed as a parameter.

NextInSet This function is the equivalent of **NextRecord**, except that it works on the current Set instead of the entire index. **NextInSet** is limited to working on index files.

PreviousInSet This function is the equivalent of **PreviousRecord**, except that it works on the current Set instead of the entire index. **PreviousInSet** is limited to working on index files.

ISAM Parameters

The ISAM parameters specify the characteristics of the data files and indices used in your application program. The parameters may be stored in a regular ASCII text file (called the ISAM parameter file) and/or structures defined in your application code. If your application uses the high level ISAM routines, you must construct an ISAM parameter file and/or use the special incremental ISAM structures defined later. Any text editor can be used to create or modify ISAM parameter files (provided special control characters are not embedded in the parameter file).

ISAM File Relationships

The ISAM routines assume that the relationship among the data and index files is hierarchical in the sense that each data file may have many indices, while each index relates to only one data file.

For example, an accounts payable system may define a vendor data file indexed by vendor name and number, and an invoice data file indexed by vendor number and invoice number. In this case each of the data files has two indices. Even though each of the data files is indexed by vendor number, there is a separate vendor number index for each data file.

The ISAM routines also assume that the key values for each index can be derived from the contents of the data records of the associated data file. As described below, the Key Segment parameter specifies how to extract a key value from a data record. See *ctvxmg.c* for a parameter-file-based example of variable length records and their associated key values. *ctixmg.c* is the same example as *ctvxmg.c* except that it uses internal incremental ISAM structures instead of a parameter file.

With only two exceptions, c-tree requires key values to be well formed up to the full length of the key (including the optional duplicate suffix). The exceptions are:

- the **FirstInSet** and **LastInSet** functions in which you explicitly indicate the number of bytes of the target key which are significant; and the hence a complete key is not necessary; and
- key segments located in varying length fields in the latter portion of a variable length data record. In this case, c-tree automatically pads the key segment to the full segment length. Again, this is only applicable to the ISAM routines.

When using functions such as **GetGTKey**, **GetGTEKey** or **GetLTKey**, a complete key is necessary in the sense that the bytes at the end of the target which you are not concerned about still must be set to an appropriate (padded) value. In these functions, you typically ensure that the trailing insignificant bytes of the target are null (zero bytes).

When a low-level index function requires both an input key (*target*) and an output key (*idxval*), then be sure that *target* and *idxval DO NOT* point to the same location.

The **AddResource** and **AddVRecord** ISAM routines automatically form complete values from the contents of the data records passed to them. The low level **AddKey** function must be passed a complete key.

c-tree makes no assumptions about the individual bytes comprising a key value. In particular, a null byte IS NOT interpreted as the end of a key value. However, the varying length field concept supported for variable length data records under ISAM control requires the concept of a field delimiter. The field delimiter separates fields

in the latter portion of a variable length data record. Therefore, a key value located in a varying length field cannot contain the field delimiter byte.

ISAM Parameter File

This section deals with the ISAM Parameter File. Incremental ISAM structures are addressed in the next section.

ISAM Parameter File Organization

The ISAM parameter file is comprised of five types of records:

- Initialization Record
- Data File Description Records
- Index File Description Records
- Optional Index Member Records
- Key Segment Description Records

Each parameter file begins with one Initialization Record which specifies, among other things, the number of data files. Then for each data file there are a group of records beginning with a Data File Description record. Each Data File Description record specifies the number of indices it uses. For each of these indices there is a group of records beginning with the Index File Description record or Index Member record. Each Index File Description record and/or optional Index Member record is followed by one or more Key Segment Description records.

The overall organization of the records is shown in the following schematic and reflects the hierarchical relationship among the data files and their corresponding indices:

PARAMETER FILE ORGANIZATION

```
Initialization Record

   Data File Description Record
     First Index File Description Record
         First Key Segment Description Record
         :::
         Last Key Segment Record

     Optional Index Member Records
         First Key Segment Description Record
         :::
         Last Key Segment Record
     :::
```

```
Last Index File or Index Member Record
    First Key Segment Description Record
    :::
    Last Key Segment Record

REPEAT FOR EACH DATA FILE
```

Parameter File Contents

Each of the fields comprising the parameter file records is separated from neighboring fields by one or more "white space" characters (blanks, tabs, or new lines). In the following subsections, if a parameter corresponds to a formal parameter in a c-tree function call, its name is included after the parameter's brief description (e.g., *bufs* in the first position of the initialization record).

Initialization Record

The Initialization Record is made up of four numeric fields:

- number of index file buffers (*bufs*)
- number of indices
- number of node sectors (*sect*)
- number of data files

The first and third parameters correspond to the first and third parameters of the **InitCTree** function (*bufs* and *sect*) described earlier. For each index specified in the second parameter, there is either an Index Description record or an Index Member record. For each data file specified in the fourth parameter there is a Data File Description record. Note that the Number of Node Sectors times 128 determines the size of the B-Tree nodes.

NOTE: the number of data files specified by the fourth parameter of the Initialization Record must match exactly the number of Data File Description Records. However, the number of index files given by the second parameter can be equal to OR GREATER than the actual number of indices. If it is greater, then c-tree will reserve extra file control blocks which your application program may utilize with its own calls to **OpenFile**, **OpenIFile** or for use in more flexible numbering of the files in the parameter file.

Data File Description Record

Each Data File Description record is made up of six required fields and two optional r-tree fields:

- data file number (*datno*)
- data file name (*filnam*)

- record length (*datlen*)
- file size extension parameter (*xtdsiz*)
- file mode (*filmod*)
- number of associated index files

If RTREE is defined in *ctoptn.h*, then each Data File Description record must have two additional fields:

- symbolic name (without any spaces) of the first field in the data record
- symbolic name of the last field in the data record

The first five fields correspond to the parameters of the **CreateDataFile** function. The data file name must agree with your operating system's file naming conventions. Usually it can include a disk drive designation and/or a directory path name. The record length parameter is used only when the file is created. However, it should be properly maintained since the Rebuild Utility verifies that the parameter file value matches the value in the data file header record. Recall that the *filmod* parameter specifies the multi-user access to the file, whether the file is opened permanently or virtually, whether (at create time) the data file will have fixed or variable length records, and the level of transaction processing desired (if using the FairCom Server).

The sixth parameter specifies how many indices will be maintained for the data file. Each such index will either be a separate c-tree file or a member of an index file. (See **CreateIndexFile** and **CreateIndexMember**.) If RTREE is defined in *ctoptn.h*, then the seventh and eighth parameters must be present. If, in fact, the parameter file is to be used in conjunction with r-tree, then these parameters must match actual entries in the data object definition array (DODA), otherwise r-tree will report an error. (The DODA is described in the r-tree Programmer's Guide.) If the parameter file is not actually for use with r-tree, then any arbitrary symbolic names may be used as long as there are no embedded blanks.

Index File Description Record

Each Index File record is comprised of eleven required fields and one optional r-tree field:

- index file number (*keyno*)
- index file name (*filnam*)
- key length (*keylen*)
- key type (*keytyp*)
- duplicate flag (*keydup*)
- number of additional index file members (*nomemb*)

- file size extension parameter (*xtdsiz*)
- file mode (*filmod*)
- null key flag
- empty character
- number of key segments

If RTREE is defined in *ctoptn.h*, then one additional parameter is required:

- symbolic index name

The first eight fields correspond to the parameters of the **CreateIndexFile** function. The index file name must agree with your operating system's file naming conventions. Usually, the index name may contain a disk drive designation and/or a directory path name. The third through seventh parameters are used at the time the index file is created. However, they should be maintained carefully since the Rebuild Utility verifies that the ISAM parameter file values agree with the corresponding values stored in the index file header record.

Remember, if duplicates are allowed, the last four bytes of the key value are reserved for the four bytes of the associated data record position. For example, a key length of twelve bytes combined with duplicate values results in eight bytes for the actual key value and four bytes for the tie-breaking data record position.

Skipping the ninth and tenth parameters for now (see Null Key Values below), the eleventh parameter specifies the number of segments which will be concatenated to form a key value from the data record contents.

If RTREE is defined in *ctoptn.h*, then the twelfth parameter is required. It is a symbolic name for the index file. It should start with an alphabetic character and may include upper and lower case letters, numbers and under scores. It may not include embedded blanks. Unlike the field names in the Data Description record, these do not and must not match entries in the r-tree DODA. (Symbolic names for the data files are not necessary because each data file has a unique name.)

Null Key Values

The ninth parameter (null key flag) of the Index File Description record determines if null or missing key values will be detected. After the key segments have been concatenated according to the Key Segment Description records, the null key flag parameter determines whether or not to test for a missing value. If the null key flag parameter is zero, no check for a null or missing value is made. If the null key flag is set to one, then the resulting key value is compared with the empty character (tenth parameter). If the key value is composed entirely of empty characters, then the key value is considered null or missing and is not entered into the index file.

The empty character is expressed in the decimal equivalent of the character. For example, an ASCII space (20 hex) is specified with a value of 32. A null byte (0 hex) is specified with a value of zero.

Optional Index Member Record

It is not necessary for each index associated with a data file to be in a separate file. Indices can be combined in the same physical file. If the *nomemb* parameter of an Index File Description Record is greater than zero, then the following *nomemb* Index File Description Records must be replaced by *nomemb* Index Member Records. For example, if a data file has three indices, and if the second index has one additional member (*nomemb* equals one), then there will be a total of two Index Description Records and one Index Member Record.

The Index Member Record is composed of seven required fields and one optional r-tree field:

- key number (*keyno*)
- key length (*keylen*)
- key type (*keytyp*)
- duplicate flag (*keydup*)
- null key flag
- empty character
- number of key segments

If RTREE is defined in *ctoptn.h*, then each Index Member record has one additional parameter:

- symbolic index name

These fields have the same interpretation as the corresponding fields in the Index Description Record.

NOTE: The Index Member records do not include explicit member numbers (*membno*). The sequence of Index Member records following an Index File Description record determines the member number of each Index Member (the first Index Member record is member number one, and so on). While a key number (*keyno*) is specified for each member, there is no freedom in the *keyno* value. The *keyno* specified in each Index Description Record must equal the *keyno* value of the host index file plus the member number or else error IMKY_ERR will be raised in **CreateISAM** and **OpenISAM**.

Key Segment Description Record

Each key value is composed of one or more segments. A segment is simply a sequence of bytes from specified offset within the data record. Each Key Segment Description record is comprised of three fields:

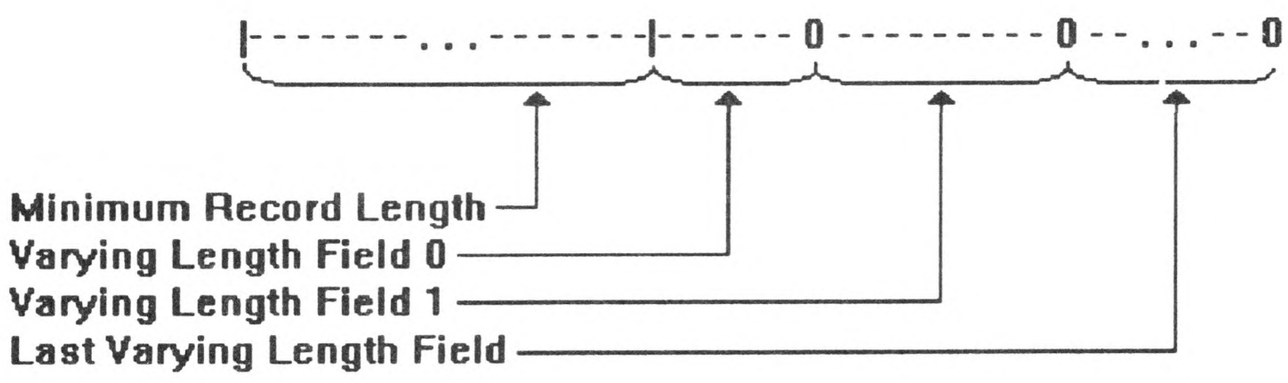
- segment position
- segment length
- segment mode

Segment Position

The segment position specifies the location of the key segment within the data record. This can be specified in one of three ways, as determined by the *segment mode* value (as discussed later);

1. **Absolute byte offset**. The number of bytes from the beginning of the data record to the first byte of the key segment. The smallest offset (zero) implies that the segment begins with the first byte of the record. If a segment begins with the *nth* byte of the record, its offset value should be *n minus one*.

2. **Relative field number**. The segment position is treated as a field number, not a byte offset. This applies only if you are using variable length data records, and all of the fields in the variable length portion are themselves variable length fields terminated with a delimiter. This delimiter is typically the NULL character, unless you have changed the value of the delimiter with the **SetVariableBytes** function. A segment position of zero corresponds to the first variable length field, which begins at the first byte beyond the variable length files's minimum record length. (Remember, when a variable length file is created, the record length parameter is treated as the minimum record length which corresponds to the fixed length portion of the file.) The following schematic demonstrates how varying length fields are numbered (the delimiter is shown as a null byte, but it can be any delimiter that you have set):



3. **Record Schema field number**. If you are using a Record Schema (see page D-1) then you can specify the segment position by the field number in the record schema. The first field in the schema is field zero.

Segment Length

The segment length specifies the number of bytes that this key segment will take in the key itself. The sum of the segment lengths for an index file must equal the key length unless duplicate keys are allowed. In this case the segments should sum to

the key length less four bytes. If the sum of the segment lengths does not match the key length, **CreateISAM**, **OpenISAM**, **CreateIFile**, or **OpenIFile** will return error ISLN_ERR.

c-tree will automatically pad a key segment to the full segment length using the padding byte. This generally defaults to an ASCII space (20 hex), but that can be changed by using the **SetVariableBytes** function. If you are using a field other than a variable length field, you should try to make the segment length less than or equal to the actual size of the field.

Segment Mode

The segment mode determines what, if any, transformations must be made to the key segment. In addition, it determines which of the three methods of determining the segment position will be used.

Detailed information on key segment modes is presented in the following section. If you are planning on using a Record Schema to manage the building of your keys, please refer to page D-1 for more information.

Key Segment Modes

As discussed before, an ISAM key is composed of one or more key segments. The Key Segment Mode value is used to tell c-tree how to locate the information necessary to create the key segment (along with the segment position and segment length) and how to translate it, if necessary.

The following table lists the valid key segment mode values. When you are creating a Key Segment Description record in an ISAM parameter file, you will use the *mode value*. The *segment position interpretation* column tells you which of the three segment position types this mode represents. The *symbolic constant* can be used if you are building an Incremental ISAM ISEG structure (as explained on page 5-27.

Mode Value	Symbolic Constant	Segment Position Inter-pretation	Explanation
0	REGSEG	Abs. byte offset	no transformation.
1	INTSEG	Abs. byte offset	unsigned integer.
2	UREGSEG	Abs. byte offset	lower case letters are converted to upper case.
3	SRLSEG	Abs. byte offset	automatic 4 byte sequence number.
4	VARSEG	Relative field #	no transformation, pad to length.
5	UVARSEG	Relative field #	lower case letters are converted to upper case, pad to length.

Mode Value	Symbolic Constant	Segment Position Interpretation	Explanation
6	YOURSEG1		reserved for your use, contact FairCom for the creation of custom Servers which include your special segments
7	YOURSEG2		reserved for your use, contact FairCom for the creation of custom Servers which include your special segments
8	SGNSEG	Abs. byte offset	signed integer.
9	FLTSEG	Abs. byte offset	floating point (float or double).
10	DECSEG	Abs. byte offset	scaled BCD.
11	BCDSEG	Abs. byte offset	unscaled BCD.
12	SCHSEG	Schema field number	will transform according to the underlying data type, padding varying length string fields.
13	USCHSEG	Schema field number	will transform according to the underlying data type, lower case letters are converted to upper case, padding varying length string fields.
14	VSCHSEG	Schema field number	will transform according to the underlying data type, padding fixed or varying length string fields.
15	UVSCHSEG	Schema field number	will transform according to the underlying data type, lower case letters are converted to upper case, padding fixed or varying length string fields.

Don't be overwhelmed by the number of key segment modes. For fixed length records there are just a few variations. If you are not sure what to use, start off with REGSEG (0). This takes the value just as it is found in the record, at a given position. If the data field is an unsigned integer, signed integer, floating point or a BCD number, use the appropriate key segment mode. c-tree will do whatever transformations that are necessary to make the key sort in the appropriate order. Another basic mode is UREGSEG (2), which will translate lower case letters into upper case letters. If you look at the ASCII collating sequence you will see that all lower case letters come after all upper case letters. If you had "ABC", "DEF" and "bcd" as key values, the sequence they would sort in would be:

```
ABC  DEF  bcd
```

Most likely this is not the order that you want the keys in. By using UREGSEG the lower case letters will be translated to upper case, so the sequence would be:

```
ABC  BCD  DEF
```

Fields in the variable length portion of a variable length record must be handled a little differently. Since we cannot predict an absolute byte offset for these fields, we have to be able to calculate a position by counting field delimiters. Use VARSEG (4) or UVARSEG (5). Since the key itself must be a particular length, we will pad it to the proper length.

Key segment modes 12 and up relate to Record Schemas, and are described in Appendix D.

Preparing Key for Key Searches

c-tree stores the key segment in the index *AFTER* it is transformed according to the segment mode. It is essential that all target key values be transformed prior to key value searches. You can use the **TransformKey** function to accomplish the transformations. If you use an **extended function call** (such as **InitISAMXtd**) to initialize c-tree, **TransformKey** will automatically be called for each ISAM search routine, unless you indicate that you do not want the automatic transformation via the user profile mask.

Sequence Number Segments

Segment mode 3 (SRLSEG) is unique since it is the only segment mode which places data in your data record. All the other modes simply cause data to be extracted from your record, optionally transformed, and then made part of a key value. The automatic sequence number feature of c-tree permits your application data record to contain an increasing (or decreasing as explained below) 4 byte data field. Each time a new record is added to the data file, c-tree increments the data file's serial (or sequence) number. Segment mode 3 causes this sequence number to be placed in your record structure at the byte offset indicated by the key segment information.

More than one key may include a key segment with mode value 3, but each of the segments with mode value three must have an identical byte offset. That is, the different keys using the serial number will share the same 4 byte region of the data record.

If you wish the serial number to be updated each time a record is rewritten, then you must delete the record and then add the new record. Using **ReWriteRecord** function causes the serial number to remain the same.

Descending Key Segment Values

Any of the key segment modes presented in the previous section can be modified to force the key segment to be collated in **descending** (instead of ascending) order by adding a value of 16 to the mode (or "OR"ing the mode with the symbolic constant DSCSEG).

For example, a segment mode of two will cause the following key values to be sorted in ascending order:

```
"ABC"
"BCD"
"DEF"
```

A segment mode of 18 (2 + 16) will cause the key values to be stored in **descending** order:

```
"DEF"
"BCD"
"ABC"
```

Because this modification is performed at the segment level, it is possible to have some segments collating in ascending order while others collate in descending order in the same key value. For example, you may have a customer order key stored in increasing order by customer number, while orders for the same customer are ordered in descending order by date. This would make the most current orders come first for each customer.

Remember the information about preparing keys for searches that is discussed on page 5-18.

Alternative Collating Sequence

Any of the key segment modes can be modified to use an alternative collating sequence by adding 32 to the mode (or "OR"ing the mode with the symbolic value ALTSEG). For example, a mode of 32 (0 + 32) will cause the key segment to be stored in an other than the standard ASCII collating sequence. This is especially useful for extended character sets associated with languages which do not collate correctly according to the standard ASCII sequences.

To be able to use an alternate collating sequence you must have Resources enabled (resources are explained on page 8-1). After you have created your index file you will use the function **SetAlternateSequence** to place the alternate collating sequence as a resource in that file. Any key segment that is set to use an alternate sequence will use the one set for that index file.

> **If you change the alternate collating sequence for an index file that already containts key values you must rebuild the index, or you will not be able to predict the sequence that the keys will appear in.**

The function **GetAlternateSequence** can be used to bring the alternate collating sequence into memory so that you can examine it.

Database Examples

On your disk you will find six files which make up two complete database examples using the c-tree ISAM routines:

- *ctexam.p* - an ISAM parameter file using fixed length records
- *ctvxam.p* - an ISAM parameter file using variable length records
- *ctexmc.c* - a simple program to create the ISAM files
- *ctexmg.c* - a program to add, update and scan the fixed length files
- *ctvxmg.c* - a program to add, update and scan the variable length files
- *ctixmg.c* - the same program as *ctvxmg.c* except that it uses incremental ISAM files instead of an ISAM parameter file, and includes automatic rebuilding.

The example databases maintain a customer file indexed by name, number and Zip Code. The names are maintained alphabetically by the first fourteen bytes of the last name and the first six bytes of the first name and an automatic sequence number. The customer numbers are handled as four byte integers. The Zip Code index is based on the nine characters of the Zip Code and the first nine bytes of the last name and the same automatic sequence number. This yields an alphabetical ordering identical Zip Codes.

Fixed Length Parameter File Examples

The contents of *ctexam.p* will be similar to those shown below.

ISAM PARAMETER FILE FOR FIXED RECORD LENGTH

```
18 3 4 1

3     cust.dat    128  4096  1  3
 0    custnam.idx  24 0 0  1  4096  1 1  32  3
   6  14 2
  30   6 2
 121   4 3
1                      4 0 0                  0   0  1
   2   4 1
2    custzip.idx  22 0 0  0  4096  1 1  32  3
 112   9 2
   6   9 2
 121   4 3
```

This parameter file is interpreted as follows. There will be eighteen index file buffers, up to three index files, 512 byte B-Tree nodes, and exactly one data file.

The first index is assigned key number 0, and is in the file named CUSTNAM.IDX which will be incremented in 4,096 byte chunks. The keys are unique, 24-bytes long and they are scanned left-to-right. The keys have three segments: 14 bytes, 6 bytes and 4 bytes long. The first two segments are converted to upper case, and are made up of the last name and first, respectively. The third segment is based on the automatic sequence number maintained by c-tree for each data file. The sequence number ensures that keys which are otherwise the same will be stored in order of entry in the data base.

NOTE: c-tree will automatically place the sequence number in the record at byte offset 121 as specified by the third segment's offset value.

The second index is assigned key number 1, and is the first (and only) additional member of the CUSTNAM.IDX file. The keys are unique, 4-byte values. Keys are not checked for missing values (i.e., all data records will generate a key value). The key is comprised of one segment treated as an unsigned integer.

The last index is assigned key number 2, is named CUSTZIP.IDX and has 22 byte unique keys. The key segments are made up of the nine character Zip Code, the first nine bytes of the last name, and the same automatic sequence number used in the name index.

NOTE: If more than one index uses the automatic sequence number, then all such indices must use the same sequence number. Otherwise, the results will be unpredictable. Also, the segment length of the sequence number must be 4.

NOTE: Each data file, index file and index file member is assigned a unique number. This is the number by which the data file and indices are referenced in the c-tree functions. Note also that the additional index members are numbered consecutively based on the host key number.

If RTREE is enabled in *ctoptn.h*, then the preceding parameter file might look as follows:

ISAM PARAMETER FILE WITH RTREE ENABLED

```
18 3 4 1

   3   cust.dat    128  4096  1  3 delflag cust_last
    0  custnam.idx  24 0 0  1  4096  1 1  32  3 name_key
       6  14 2
      30   6 2
     121   4 3
    1                  4 0 0              0   0  1 numb_key
       2   4 1
    2  custzip.idx  22 0 0  0  4096  1 1  32  3 zipc_key
     112   9 2
```

```
   6    9 2
 121    4 3
```

Variable Length Parameter File Example

The example parameter file for the variable length example takes advantage of the key compression and the varying length fields supported by c-tree.

ctvxmg.c contains the source for the variable length example, and *ctvxam.p* is its associated parameter file. The contents of *ctvxam.p* will be similar to those shown below (assuming that RTREE is not enabled).

```
ISAM PARAMETER FILE FOR VARIABLE RECORD LENGTH
18 3 4 1

3         vcust.dat      15  4096  5  3
 0     vcust.idx  24   4 1  2  4096  1 0  32  2
     0  15 5
     4   5 2
 1                      4  0 0                0   0  1
     0   4 1
 2                    22 12 1                1  32  2
     4   9 2
     0   9 5
```

The important points to note in this example are:

- the last name index (keyno = 0) has a key type of 4 which means leading character compression will take place. Padding compression is not used since the second segment of the key is based on the Zip Code and will almost always be the full length.
- the first segment of the last name index has a mode value of 5: the segment is found in the zeroth varying length field (i.e., right after the minimum record length portion of each data record), not at absolute byte position zero. It will be padded, if necessary, to a full 15 bytes and converted to upper case.
- the Zip Code index (keyno = 2) has a key type of 12 which means both leading character and padding compression will take place. Since the second segment of this index is based on the last name, there is a good chance that many of the keys will have padding.

The main differences between *ctvxam.p* and *ctexam.p* are:

- the data file uses variable length records (file mode of 5 implies virtually opened, variable length records and file sharing in multi-user systems) with a minimum record length of 15 bytes;

- all the indices have been combined into one physical index file;
- the key segment offsets are modified since all the fixed length fields have been moved to the beginning of the record structure.

In these examples we have used a non-zero file size extension parameter. A zero file size extension parameter implies that the file will grow one record at a time instead of growing in large chunks. The advantage of growing in chunks is that every time the file is extended, c-tree forces the directory entries associated with the file to be updated.

Multiple Data File Parameter Setup

The ISAM parameter file can accommodate more than one data file. The following data show how a parameter file would look for two data files, each of which have two index files. Note that the indentation is ignored by the programs, but is helpful in reading the material.

```
24 4 4 2
   0 VENDOR.DAT  384  8192 0  2
     2  VENDOR.IDX  14  0  1  1 8192 0 1  32  1
          0 10 0
     3                6  0  0           0   0  1
         12  6 0
   1 INVOICE.DAT 192  8192 0  2
     4  INVOICE.IDX 16  0  1  1 8192 0 0   0  2
         32  6 0
          0  6 0
     5               10  0  1           1  32  1
          0  6 0
```

The application program which uses this parameter file would refer to the vendor data file (VENDOR.DAT) as file number 0, and the invoice data file (INVOICE.DAT) as file number 1. Each data file has two indices. In both cases the indices are store in one physical file.

Notice that except for key number 3 which does not accept duplicates, all the segment lengths sum to four bytes less than the actual key lengths to accommodate the suffixes.

Key Value Assembly

The ISAM parameters for your application can specify that key segments are to be translated in some manner before being concatenated and added to your index files. However, c-tree does not automatically translate your target keys in the same way. Any ISAM function that is passed a target key (such as **FirstInSet**) expects a fully and properly formed key. c-tree provides you with several ways to create such a key.

TransformKey

TransformKey will convert a target key according to your ISAM parameters for a given index file. For example, if you have reversed the bytes of a key segment (segment mode equal to one), then passing a target key to **TransformKey** will cause the appropriate segment of the target key to be reversed.

TransformKey expects that you have concatenated all the necessary segments into a target key pointed to by the target parameter. Since **TransformKey** constructs the translated key in place, you must ensure that the key area is at least as large the key length (including suffix) defined for the index file.

CurrentISAMKey

TransformKey expects a single target value that consists of all of the key components already concatenated together. Sometimes this can be difficult (or at least inconvenient) to construct. The function **CurrentISAMKey** is provided to return a key for the current ISAM record.

This function returns a pointer to an already assembled key value for the specified index file from the current ISAM record.

Incremental ISAM Structures

The incremental ISAM structures permit your application program to create (**CreateIFile**), open (**OpenIFile**, **OpenFileWithResource**), close (**CloseIFile**, **CloseRFile**) and rebuild (**RebuildIFile**) individual ISAM files (actually data file / index file pairs) independent of an ISAM parameter file. The incremental ISAM structures contain essentially the same information as the parameter file, but it permits the information to be stored in your application instead of an external file, and it permits direct control of the opening and closing of individual files instead of a complete set of files.

Incremental file control can be accomplished whether or not a parameter file is currently active. However, closing a parameter file also closes all the currently opened ISAM files including those opened via the incremental ISAM open.

There are three structures which work together to enable the incremental ISAM capability: IFIL which defines the data file, IIDX which defines the indices, and ISEG which defines the key segments. These structures are defined in the *ctifil.h* header file which should be included in any application program which utilizes the incremental ISAM functions.

IFIL Structure

All the incremental ISAM functions require a pointer to an IFIL structure as their input parameter. The IFIL structure defines the characteristics of the data file and includes a pointer to a structure which defines the associated indices. The formal type definition for IFIL follows:

```
typedef struct ifil {
   TEXT     *pfilnam;  /* file name (w/o ext)    */
   COUNT     dfilno;   /* data file number       */
   COUNT     dreclen;  /* data record length     */
   UCOUNT    dxtdsiz;  /* data file ext size     */
   COUNT     dfilmod;  /* data file mode         */
   COUNT     dnumidx;  /* number of indices      */
   UCOUNT    ixtdsiz;  /* index file ext size    */
   COUNT     ifilmod;  /* index file mode        */
   IIDX     *ix;       /* index information      */
   TEXT     *rfstfld;  /* r-tree 1st fld name    */
   TEXT     *rlstfld;  /* r-tree last fld name   */
   COUNT     tfilno;   /* temporary file number */
} IFIL;
```

Corresponding IFIL Member	c-tree Parameter	Interpretation
`pfilnam`	*filnam*	The file name pointed to by pfilnam is shared by both the data file and index file. The name should not include the file extent since ".dat" and ."idx" (as defined in *ctifil.h*) will be added to the name for the data file and index file (the extensions can be changed by using *extended function calls*.
`dfilno`	*datno*	The file numbers assigned to the data file and any associated indices run consecutively. If dfilno is negative, then c-tree finds the first available block of numbers. If dfilno is positive, then c-tree assigns dfilno as the data file number. In either case, *tfilno* is set equal to the data file number.
`dreclen`	*datlen*	data record length
`dxtdsiz`	*xtdsiz*	data file extension size
`dfilmod`	*filmod*	data file mode indicator
`dnumidx`	N/A	The number of associated indices (which may be zero).

Corresponding IFIL Member	c-tree Parameter	Interpretation
ixtdsiz	*xtdsiz*	index file extension size
ifilmod	*filmod*	index file mode indicator
ix	N/A	Pointer to IIDX index structures as defined below.
rfstfld	N/A	An optional pointer to the symbolic name of the first data field in the record as specified in the r-tree DODA.
rlstfld	N/A	An optional pointer to the symbolic name of the last data field in the record.
tfilno	N/A	Output parameter set equal to the file number assigned to the data file.

IIDX Structure

Each index is described by an IIDX structure. If a data file has associated indices, then the ix member of the IFIL structure must point to an array of IIDX structures. In particular, there must be dnumidx elements (i.e., IIDX structures) in the array. The formal type definition for IIDX follows:

```
typedef struct iidx {
   COUNT ikeylen,  /* key length              */
         ikeytyp,  /* key type                */
         ikeydup,  /* duplicate flag          */
         inulkey,  /* null key flag           */
         iempchr,  /* empty character         */
         inumseg;  /* number of segments      */
   ISEG      *seg;     /* segment information   */
   TEXT      *ridxnam; /* r-tree symbolic name  */
} IIDX;
```

Corresponding IIDX Member	c-tree Parameter,	
ikeylen	*keylen*	key length
ikeytyp	*keytyp*	key type
ikeydup	*keydup*	duplicate flag
inulkey	N/A	null key flag
iempchr	N/A	empty character
inumseg	N/A	number of key segments

Corresponding IIDX Member	c-tree Parameter,	
`seg`	N/A	pointer to ISEG key segment structures
`ridxnam`	N/A	pointer to optional symbolic index name for use with r-tree

ISEG Structure

Each key segment is described by an ISEG structure. The seg member of each index IIDX structure must point to an array of ISEG structures. In particular, there must be inumseg elements (i.e., ISEG structures) in the array. The formal type definition for ISEG follows:

```
typedef struct iseg {
   COUNT soffset,  /* segment offset   */
         slength,  /* segment length   */
         segmode;  /* segment mode     */
} ISEG;
```

segmode can be specified by using a key segment mode number (i.e. 0, 1, 2) or a symbolic constant as shown in the table on page 5-16 (i.e. REGSEG, INTSEG, UREGSEG).

Incremental ISAM Examples

The following C declarations show how the variable length data base example shown before can be expressed in terms of initialized incremental ISAM structures. Because of the nested relationship of IFIL, IIDX and ISEG, we start off by defining the segment information, then the index information, and then the data file information. For the sake of completeness, we have added the optional r-tree symbolic names.

```
#include "ctifil.h"

ISEG vc_seg[] = {
   {0,15,5},       /* 1st segment, 1st index      */
   {4, 5,2},       /* 2nd segment, 1st index      */
   {0, 4,0},       /* 1st segment, 2nd index      */
   {4, 9,2},       /* 1st segment, 3rd index      */
   {0, 9,5}        /* 2nd segment, 3rd index      */
};

IIDX vc_idx[] = {
   {24,                    /* key length, 1st index      */
    12,                    /* key type                   */
```

```
    1,                  /* duplicate flag on          */
    1,                  /* null key flag on           */
   32,                  /* empty char (decimal)       */
    2,                  /* number of key segments     */
    &vc_seg[0],         /* pointer to segment array   */
    "name_key"},        /* index name                 */

  { 4,                  /* key length, 2nd index      */
    1,                  /* key type                   */
    0,                  /* duplicate flag off         */
    0,                  /* null key flag off          */
    0,                  /* empty char ignored         */
    1,                  /* number of key segments     */
    &vc_seg[2],         /* pointer to segment array   */
    "numb_key"},        /* index name                 */

    {22,                /* key length, 3rd index      */
     12,                /* key type                   */
      1,                /* duplicate flag on          */
      1,                /* null key flag on           */
     32,                /* empty char (decimal)       */
      2,                /* number of key segments     */
      &vc_seg[3],       /* pointer to segment array   */
      "zipc_key"}       /* index name                 */
};

IFIL vc_dat = {
    "vcust",            /* root of file name          */
    1,                  /* data file number           */
    15,                 /* minimum record length      */
    4096,               /* data file extension size   */
    5,                  /* data file mode             */
    3,                  /* number of indices          */
    4096,               /* index file extension size  */
    1,                  /* index file mode            */
    vc_idx,             /* pointer to index array     */
    "vcust_1st",        /* 1st data field             */
    "vcust_last"        /* last data field            */
                        /* null: tfilno               */
};
```

With the above structures, the incremental ISAM functions (**CreateIFile**, **OpenIFile**, **CloseIFile**, **RebuildIFile**) would be called with a single input parameter: *&vc_dat*. *&vc_dat* is a pointer to an IFIL structure which in turn points to the array of index structures and segment structures.

It is not necessary to put all the segment values in one array. You could use separate arrays of ISEG structures for each index. (However, it is true that all indices for a single data file must be in a single array of IIDX structures.)

Incremental ISAM and Resources

If Resources are enabled when you create a file with **CreateIFile**, the information stored in the IFIL structures will be stored as a resource in the data file. You can use the **OpenFileWithResource** and **CloseRFile** functions to open and close the file(s) without having to refer to a specific IFIL structure in memory. You can assign the file numbers via a parameter, or ask c-tree to assign appropriate file numbers.

6. Multi User Concepts

File Locking

When c-tree Plus is used in a network or multi-tasking environment, it is necessary to specify whether individual files are shared among the users, or opened exclusively for one user.

Multi-User File Mode

When opening a file you can specify the type of shared user access by including the appropriate value in the *filmod* parameter.

The file modes available are EXCLUSIVE, SHARED and READFIL, which can be used with the other file mode parameters. An EXCLUSIVE file can only be opened by one user; a SHARED file can be opened by multiple users for concurrent read/write access; and a READFIL can be opened by multiple users for read only access. Two additional file modes, CHECKLOCK and CHECKREAD, affect **record** locking, but are specified at the time the file is opened or created in addition (optionally) to SHARED file mode. These modes are available only if you use the FairCom Server, and are discussed in the section starting on page 10-2.

EXCLUSIVE

If a file is opened with a *filmod* of EXCLUSIVE, that file can only be opened by one user. If another user already has the file open in any mode, this user cannot open the file as EXCLUSIVE. Once this user has the file opened as EXCLUSIVE, no other user can open it.

SHARED

Opening a file with a *filmod* of SHARED allows multiple users to have both read and write access to the file. If a user tries to open a file as SHARED and another user has the file open as EXCLUSIVE or READFIL, this user cannot open the file. Once this user has a file open as SHARED, other users can open the file as SHARED but not as EXCLUSIVE or READFIL.

READFIL

Use this mode if you wish to have multiple users read the file, but you want to prevent them from being able to update the file. READFIL will work only if the other users all are using the mode of READFIL..

Record Locking

Record locking controls which user is going to be able to access and update a given record. When multiple users are accessing a data file in SHARED file mode you must use record locking to prevent several users from trying to update the same record at the same time.

For low level functions you will use the **LockData** function. In ISAM functions you will use the **LockISAM** function.

In c-tree Plus we define two types of record locks; **write locks** and **read locks**.

Write Locks

A write lock can be considered to be an exclusive lock. Only one user can have a write lock on a given record. A write lock also prevents anyone from getting a read lock on the same record. This is used when a user wants to update a record, and wants to prevent any other user from updating at the same time. This is done with the record lock mode of ENABLE.

Read Locks

A read lock can be considered to be a shared lock. Any number of users can have a read lock on a record simultaneously. A read lock on a record will prevent another user from getting a write lock on this record. You would use this to allow one or more users to look at a record while ensuring that it cannot be changed or deleted by another user while they are looking at it. This is done with a record lock mode of READREC.

ISAM LEVEL Data Record Locks

The design of **LockISAM** is intended to enforce the classic two-phase locking protocol which helps ensure that multiple users cause consistent updates. Briefly, the two-phase protocol states that a transaction should never unlock a data record until it is done acquiring all the locks it needs for the entire transaction. In other words, there is a phase during which a transaction acquires record locks and does not free any locks; and a second phase during which the transaction relinquishes locks and does not acquire any new locks.

When we refer to a *transaction* in this discussion we are NOT referring specifically to Transaction Processing as will be discussing in the section on the FairCom Server. We are talking about any group of related actions when accessing a database. The concept discussed here, however, does apply to Transaction Processing as well.

ENABLE

With **LockISAM**, the first phase is initiated when **LockISAM** is called with the ENABLE parameter. Then all subsequent record operations first acquire a write lock before proceeding. As you can see you will only need to do one **LockISAM** call with the ENABLE parameter to lock a number of records.

If a subsequent call cannot get an appropriate lock on the record, that function will return a value of DLOK_ERR.

ENABLE_BLK

If you use ENABLE, and a lock cannot be obtained, the calling function will return with an error code. ENABLE_BLK is the same as ENABLE, but a process unable to get a write lock is "put to sleep" until the lock is available. Use this with caution.

READREC By using a mode of READREC instead of ENABLE you can specify read locks. You can also switch from ENABLE to READREC by another call to **LockISAM** using the READREC mode.

FREE The second phase is essentially one operation which occurs when **LockISAM** is called with the FREE parameter. All ISAM record locks currently held by this application will be released, again with just a single function call.

RESET Calling **LockISAM** with a mode of RESET is the equivalent of making two calls to **LockISAM**, first using the mode of FREE and then the mode of ENABLE. This is of value when you are traversing the data file in key sequential order. After each update the previous locks are freed and new locks are enabled for the next transaction.

SUSPEND
RESTORE
RESTRED
RESTORE_BLK

There may be situations when, during the first phase, it is desirable to read some records **without** acquiring locks on them (because these records will not be updated). In such cases, you may temporarily disable the automatic record locks by calling **LockISAM** with the SUSPEND parameter. Then, when the application is ready to begin reading records with locks again, you call **LockISAM** with the RESTORE parameter (or RESTORE_BLK to restore ENABLE_BLK). Note that this does not violate the first phase restraint of only acquiring locks since no locks are freed in the SUSPEND mode. (If you wish to return to the locking mode with read locks active instead of write locks, then call **LockISAM** with the RESTRED parameter. See the **LockISAM** discussion in the **c-tree Plus Function Reference Guide.**)

Using LockData with LockISAM

You can use the low level locking function **LockData**, as described below, to relax the constraints of the "two-phase" protocol. Specific records can be locked or unlocked. Note, however, that a record locked with **LockData** will NOT be freed with a **LockISAM**(FREE) call.

LockISAM and Resources

Resource records (see Chapter 8) locked with a **UpdateResource** or **GetResource** function call will NOT be unlocked with a **LockISAM**(FREE) call.

Low Level Data Record Locks

Record locking controls which user is going to be able to access and update a given record. There are four basic locking modes: ENABLE, ENABLE_BLK, FREE, and READREC. These modes have the same meaning as with the **LockISAM** function.

The **LockData** function is used when using low level functions to access the database. **LockData** differs from **LockISAM** in that it affects only a single record, as specified when the **LockData** function is called.

ENABLE This places a **write lock** on a given record. Only one user can have a write lock on a given record. It also prevents another user from getting a read lock. This is used when a user wants to update a record, and wants to prevent any other user from updating at the same time. You can think of this as an exclusive record lock.

ENABLE_BLK This is the same as ENABLE, but the process will "go to sleep" until the lock is granted. With ENABLE, if a lock cannot be granted then **LockData** will return an error code DLOK_ERR. With ENABLE_BLK c-tree will wait until the other user releases the lock so that you can obtain it. This should be used with caution. If c-tree detects a dead lock situation (User A has a lock on record 1 and is waiting for a lock on record 2, while User B has a lock on record 2 and is waiting for a lock on record 1) then an error DEAD_ERR will be returned.

FREE Use this lock mode to release a lock on a specific record.

READREC This is a read lock. Any number of users can have a read lock on a file simultaneously. This ensures that no other user can acquire a write lock on the record. You can think of this as a shared record lock.

Multi-User Interference

In multi-user applications it is possible for one process to interfere with another process even with all the proper locking strategies in place. In particular, one user may cause a change in a data record during the time another process is searching an index and securing a data record lock on the same record. If this happens, c-tree detects significant changes by reporting error ITIM_ERR (160). A reproducible error 160 during ISAM searches may be an indication of improper current ISAM settings during rewrite operations.

The ITIM_ERR detects two types of changes to the record being accessed:

- the record has been deleted; or
- the key value has changed.

If error ITIM_ERR is returned by an ISAM function, then the operation should be retried. For example, if **FirstRecord** returns error 160, then either the record has just been deleted or the key which was at the beginning of the index has now been changed. In either case, your program should call **FirstRecord** again to get the current first record.

Multi-User Updates

The most significant difference between coding applications for single-user environments and multi-user environments (including local area networks) has to do with performing record updates (rewrites) and record deletions .

The basic problem is that to perform a rewrite or a deletion in a multi-user system, you must own a record lock on the record of interest. However, one must be careful to help ensure that locks are held for the shortest time possible.

There are two basic approaches to avoid prolonged data record locks:

- Do not acquire a data record lock until the user has completed all updates on the record in a local buffer. This approach requires three record buffers: one to hold the current ISAM record (for which no data record lock has been requested); a second to hold the updated record; and a third to reread the current ISAM record with a **LockISAM** request when the updates have been completed. If you cannot get the lock on this reread or if the contents of the reread buffer do not agree with the contents of the original current ISAM buffer, then the operator must be informed that the update could not be accomplished.
- Acquire a data record lock when the user reads the record to begin the update process, AND if the updates are not completed in a specified time interval, AND if the operator cannot acknowledge that updates are still desired, then release the record lock(s) and cancel the update session. This requires real time control of the data input process which may not be available.

The first of these two approaches is used in *ctexmg.c*. Examine how the change function is used to delay getting a record lock until the rewrite or delete has actually been committed.

If either of the two above approaches is not used, then the application typically uses a scheme in which the lock is acquired when the user reads the record in preparation to update the record, AND does not relinquish the lock until the update is completed: no matter how much time is required. The classic problem situation which arises with this technique is when an update is interrupted (say by the updater going to lunch). Then a lock is held for a (relatively) long time on the data record, blocking other users from update requests on the same record.

7. Sample Programs

The best way to learn how to use c-tree Plus is to examine programs that use it. We have created two series of sample applications to help you understand how to apply what we have shown you in this Guide so far.

We start with a series of very simple applications that start with a basic file maintenance program, and then build until we have a more sophisticated program. We have tried to keep the non c-tree material to a minimum. We have briefly introduced these sample programs in the beginning of Chapter 4 and Chapter 5.

The programs in the second series are more complex, and illustrate many advanced features. Some of the non c-tree material is a bit more sophisticated as well. They will take more study on your part, but the understanding that you will gain is well worth the effort. These programs are used in various places throughout this Guide.

Simple Applications

The sample application we use is a very simple one. We don't want to obscure the c-tree Plus functions too much. We are going to maintain a simple inventory file, with just one key field. We start off with just the basics that are necessary to start and create a file and index, and then we add on a variety of functions. This approach lets you study each of the concepts that we introduce without a lot of clutter from non c-tree code, and without the confusion of advanced c-tree Plus concepts in the beginning.

These programs have been written in a manner that would allow them to be used in a single user, standard multi user, or file server mode, without any change to the code. If a function is used that doesn't apply to a particular mode, such as **StopUser**, then that function is ignored. However, there are two things that have NOT been done that could be in some modes. First, in some standard multi user environments (PCNET for instance), you may need to add a dummy lock file, as explained on page 9-1. Second, we do not use any of the more sophisticated Transaction functions that are available if you use the FairCom Server.

The programs that we provide are:

- **lowlevel.c** - all we do here is to create a data file and index, to which we can add and delete records. This sample uses the **low level functions** of c-tree.
- **isam.c** - this is a program that is identical to *lowlevel.c*, except that it uses the **ISAM functions**. You will be able to see how the ISAM functions can simplify your program. This program requires the file *isam.p* as a parameter file.

- **isam1.c** - the prior sample uses ISAM functions and an ISAM parameter file. This sample is identical to *isam.c* except that it uses **Incremental ISAM structures**.
- **isam2.c** - next we build upon the first ISAM program, *isam.c*, and add the capability of listing the file to the screen. We also introduce the concept of **"sets"**.
- **isam3.c** - the last in this series, we add the ability to scan through the file to find the record that you want, and allow you to update or delete the record.

All of these examples are built upon the same program structure, and all use the same data, index, and (where appropriate) parameter files.

Simple File Maintenance

lowlevel.c is our first sample program, a simple introduction to the c-tree Plus low level functions.

Throughout this series of sample programs we are going to use the following data structure for our data file:

```
struct INVENTORY {     /* data record format          */
   COUNT delete_flag; /* delete flag byte            */
   TEXT    item[26]; /* Item ID -this is the key value*/
   TEXT    descrip[51];    /* Item Description       */
   TEXT    location[10];   /* Item Location          */
   double  quantity;       /* Quantity on hand       */
   double  cost;           /* Unit Cost              */
   TEXT    padding[23];    /* unused padding         */
   } invent;
```

We will build just one index, using the *item* field.

Note that we have allocated on field at the beginning of the record structure for the *delete flag*. It is best to set aside space for this in your structure. If you decide that you want to use a status value for each record, then you could use this same variable for the status. Just remember to not use the value of 0xff as a status value, as that represents a deleted record to c-tree.

The program mainline is the same in all of the examples of this series:

```
main (argc,argv)
int argc;
TEXT *argv[];
{
VOID create(),initial(),database(),closeit();

if (argc  1)
```

```
   create();        /* create the database files    */
else
   initial();       /* initialize the system        */

database(); /* perform database routines */
closeit();  /* close the files, and any other ending
stuff*/
}
```

If there is any value passed on the command line, we will assume that you want to *create* the data base. Otherwise we assume that the database has already been created. You could have a more sophisticated startup than this, where you try to open the files first, and if that fails then trying to create them.

After the file system has been set up, we call *database()* to perform the operations, and when done we call *closeit()* to finish up.

Initialize the system

InitCTree

Before we can do any c-tree Plus actions we must initialize the c-tree buffers with the **InitCTree** function call. This will allocate the memory for index buffers (10 in our example), data and index file structures (3 in our example), and establish the number of sectors available in each c-tree node (4 in our example).

In single user systems you can improve performance by increasing the number of index buffers (the first **InitCTree** parameter). We suggest a maximum of 6 buffers per active index. Beyond this the return in performance is very small. The only reason that you might want to use a smaller number is if you cannot afford to allocate the space. In non-server multi user systems the number of buffers should be set to 6 no matter how many indices you are using, as we will not be using local buffering.

For most situations you should leave the number of sectors per node at 4, which allocates 512 bytes per node. It is important to be consistent with this parameter, as an index created with one value cannot be used if you use a smaller value in a subsequent call to **InitCTree**.

Create and Open files

Once we have initialized c-tree Plus we can go on to either create or open the data files and indices. When using the low level functions you must explicitly create/open each file separately. The create functions are almost identical to the open functions, except that the create functions have some additional parameters that are stored in the header record of the data file or index. This is information such as the record or key length, whether the data file is to have fixed or variable length records, and so forth.

Two files are created in our example. The first file is the data file, with a fixed record length of 128 bytes. The *xtdsiz* parameter is set to 4096, which means that every time the file is extended in size it will be extended by 4096 bytes. This improves the efficiency of the system. Note that we have included a file mode of SHARED, to

allow multiple users access to the data file. The second file is the index, with a key length of 25. We are not allowing duplicate keys.

Error Handling

If an error occurs in one of the c-tree Plus functions, we call one of two error handling functions. *terminate()* is called if the error occurs at a time where we don't want to continue until the error is corrected. *func_error()* is called when the situation is not bad enough for the program to be terminated.

uerr_cod

In our example we have a very basic error handler in *func_error()*. Based on the universal error variable *uerr_cod*, we print a number of different error messages. If the error is not one that we have a specific message for, we just print the error number. You certainly can create a more sophisticated error handling routine than this.

terminate() prints an error message that we pass to it, and calls *func_error()*. The next function invoked is **StopUser()**. In single user and non-server multi user situations this function is ignored. If you are using the file server then you need to call this function before you exit the program for any reason. It tells the file server that this application is finished with c-tree services, which frees up the server resources so that they can be used by another application.

Adding Data

Now that the files are open, we can add records. The function *datadd()* is used.

The first step is to initialize the data record buffer. The only thing that is required for c-tree Plus is that the delete flag be set to a value other than 0xff. It is risky to not initialize this variable with some safe value.

The function *getfld()* is used to capture the data from the screen. As supplied there is no form of error checking. We leave this part of the program up to you.

translate key

After we have the data that we want to enter, we call *transkey()* to format the key the way we want it. This is a function that we have created in this sample program. In this case we are doing two things. First, we translate all of the alpha characters to upper case. For the purpose of our example we have decided that we don't want to distinguish between upper case and lower case characters in our keys. This will also keep the collating sequence for the keys in a usable form. The second step of the key translation is the most important. We pad the key to its full length with null characters.

Padding the key is an extremely important step. c-tree Plus keys are NOT treated as standard null-terminated C strings. The full length of the key, as specified when the index was created, is important even if the actual value used is shorter. If you are using string values as keys you must pad them to their full length, or you may

not be able to retrieve them later. For example, let's look at a situation where we have a key length of 10, and we add two key values. The first key value is "ABCDEF-GHIJ", which is 10 characters long. After we add this key to the index we are going to add another, "XYZ". This is only 3 characters long. While adding the keys we have been using the same buffer variable to store the values. If we look at what is stored in the variable when "XYZ" is there, we see that the contents of the variable are "XYZ∅EFGHIJ", where ∅ represents the null character that terminates the string. **All 10 characters are added to the index as the key!** At some later time, when we want to retrieve this record, we will have a hard time finding an exact match to the key when we enter the value "XYZ", as c-tree Plus will be looking for the rest of the information. By padding the values to their full length with some consistent character, such as a null or space, we will find that our programs will work the way we expect them to.

GetKey

Now that we have a key to work with, we want to see if the key already exists in the index. We use the **GetKey** function to look for an exact match. If the function returns a value then we have found a match, and we don't want to try to add the key again. A zero means that no match was found, and we have to check *uerr_cod* to see if there was an error or not.

NewData

The process of adding the key to the index and writing the record to the data file takes several steps. First we are going to call **NewData** to obtain a data record position to write the record at. c-tree Plus keeps a list of records that have been deleted and will reuse those records when you call **NewData**. If there are no deleted records then **NewData** will automatically extend the file to gain a new record position.

WriteData

Once we have the record position we can write the data record to the file with the **WriteData** function. If some error occurs then we will want to return the data record position to the file with the **ReleaseData** function.

AddKey

Now that the record is in the data file we can add the key to the index with the **AddKey** function. We do this after the data record was written for two reasons. First, **AddKey** requires the data record position to store along with the key value. Second, since most access to the file will be through the index, we want to be sure that the record exists before the key is written. In a multi-user environment we don't want the key to be written, and then accessed by another user before the data record was written.

LockData

Note that any time after the call to **NewData** that we leave this function, either due to a failure of some function or the successful completion of the process, we make a call to **LockData**. When **NewData** acquires a data record it puts a lock on the record that must be released before any other user can access it. If you have a single user system then the calls to **LockData** are ignored.

Delete a key

The *datdel()* function is used to delete a key from the index, and delete the record from the data file. Once again we use the *transkey()* function to translate our input key to the format that we want to use. **GetKey** is used to locate the key in the index.

ReadData

If the key is found we will use **ReadData** to read the record so that we can display it. You might decide that you want to ask the user if this is the correct record to display or not.

DeleteKey
ReleaseData

The **DeleteKey** function is used to delete the key from the index, and **ReleaseData** returns the data record to the list of deleted records.

Close the system

When we are finished with the application we must shut the c-tree Plus system and files down properly. Each open data file and index must be closed with a call to **CloseFile**. If the file has been created and/or modified during this run of the application, and we do NOT close the file properly, then the file will be considered damaged. Information is stored in the header record about the status of the file, and this is not always written to the disk until the file is closed. If you do not close the file, and the information in the header is incorrect, then the next time you try to open this file you will receive an error value that says that the file has been corrupted. You either have to throw the file away, or you can rebuild it. See the sample program *ctixmg.c* for an example of how to use the incremental ISAM function **RebuildIFile** to rebuild a file.

As we noted before in the discussion of *terminate()*, we must call the function **StopUser**() before exiting the application. This tells the file server that this user application is finished, releasing the resources for another user. If you are not using the file server then this function is ignored.

Same Application with ISAM functions

isam.c performs the identical function that *lowlevel.c* does, and will create a data file and index that matches the ones created by that program. In fact, the data and index files created by one program can be accessed by the other. The primary difference is that this program uses the ISAM functions and an ISAM parameter file instead of low level functions.

ISAM parameter file

In *lowlevel.c* we established the c-tree Plus buffer parameters with a call to **InitCTree**, and we set up the attributes of the two data files and one index by successive calls to **CreateDataFile** and **CreateIndexFile**, or to **OpenFile**. With three files to create/open, we needed four function calls.

CreateISAM
OpenISAM

With the ISAM functions we only need one call, to **CreateISAM** to create the files, or **OpenISAM** to open them if they were already created. The only parameter that is passed to either of these functions is the name of the ISAM parameter file. This parameter file contains all of the information needed to initialize c-tree Plus and to create/open the data files. In our example the ISAM parameter file is *isam.p*, as follows:

```
10 1 4 1
 0        invent.dat    128  4096  1  1 Delflag Buffer
    1     invent.idx  25 0 0  0  4096  1 0  32  1 Itemidx
       2  25 2
```

Initialization record

The first record in the ISAM parameter file is the Initialization record. This contains the same information as is used in the **InitCTree** call, although it is organized in a slightly different manner. The first value is the number of index file buffers, as before. The third value is the number of node sectors. The difference is that in **InitCTree** we had one figure for the number of data files and indices that will be opened (2 in this example), while in the ISAM parameter file we must specify the number of indices (1) as the second parameter, and the number of data files (1) as the fourth parameter.

Data Description record

Each data file must have a data description record, which is the line following the Initialization record in the ISAM parameter file shown above. The first parameter is a file number that we assign to each file (if we use a dummy lock file then that file MUST be file 0), followed by the file name, the record length, the extension size, the numeric representation of the file mode, and the number of associated index files. You should be able to match the values in *isam.p* with the values in the **CreateDataFile** calls in *lowlevel.c*, with the exception of the last one.

In our example we have prepared the data description record for use with the **r-tree** report generator by assigning symbolic names for the first and last fields in the data record. If you have set up you c-tree Plus system for use with **r-tree** then you need two names here, even if you are not going to use r-tree for the time being. The symbolic names we have used here allow us to continue using this same sample program in the r-tree Programmers Reference Guide.

Index Description record

If there is one or more indices associated with a file then there must be an index description record for each. The first parameter is the file number that we assign to the index, followed by the index file name, key length, key type, duplicate file value, number of additional index members, file extension size, and the numeric representation of the file mode. You should be able to match these with the parameters used in the **CreateIndexFile** call in *lowlevel.c*. In addition we have parameters for the null key flag and empty character (leave these as 0 and 32 respectively until you understand them fully), and the number of key segments. *Itemidx* is a symbolic name for r-tree, as discussed above.

Key Segment Description record

Each key for an ISAM index must be composed of elements drawn from the data record. You cannot have an arbitrary value from an outside source as a part of the key. Even if this wasn't a requirement of the c-tree Plus product it would make good sense, as you cannot easily rebuild a corrupted index if the key contains information that doesn't exist in the data record. In addition, keys can be created from multiple portions of the data record, and there are a number of translations that can be performed. This is specified in the Key Segment Description record.

In our example the key is a simple one of one segment. The first value is the offset of the segment in the data record. Starting at zero, the *item* field starts at byte 2 of the record (bytes 0 and 1 being the delete flag). The second value is the length of the segment, which is 25 in this case. The third value is the *segment mode*, which describes how the segment is to be translated. The chapter on the ISAM parameter file describes the various segment modes. A value of **0** means no transformation, a value of **2** (as in our example) means that the lower case letters will be translated to upper case.

Error handling

Error handling is essentially the same as before, except that we are going to look at *isam_err* instead of *uerr_cod*, and we have the variable *isam_fil* to tell us which file the error occurred with. This last is important, as ISAM functions may manage multiple files within a single call, and we need to know which file created the problem.

Adding Data

AddRecord

Adding the data to the system is very simple with the ISAM functions. After we have called the *getfld()* function to capture the data, we simply call **AddResource**(). This takes care of all of the functions performed by transkey, **AddKey**, **NewData**, **WriteData** in the low level function example.

LockISAM

Record locking is more automatic, as well. Before the call to **AddResource** we enable automatic record locking with the **LockISAM**(ENABLE) call, and later we release the lock with the **LockISAM**(FREE) call.

Delete record

GetRecord
DeleteRecord

Deleting a record and index entry is almost as simple. We use **GetRecord** to determine if the key exists (this also reads the record so that we have something to display), and then we call **DeleteRecord** to delete the key and record. Even if we had multiple indices for this file we would do just those two calls, as **DeleteRecord** will delete all keys associated with that record.

TransformKey

The key that we pass to **GetRecord** must be a properly formatted key, just as in *lowlevel.c*. Here we use **TransformKey** instead of our own function. **Transform-**

Key will take the information found in the key buffer and reformat it according to the values found in the ISAM parameter file.

Close the system

CloseISAM

Closing the data and index files is very simple. A single call to **CloseISAM** will close all of the files referenced in the ISAM parameter file. It also will call **StopUser** automatically.

Using Incremental ISAM structures

Incremental ISAM structures allow you to create, open, close, and rebuild individual ISAM files independent of an ISAM parameter file. The incremental ISAM structures contain essentially the same information as the parameter file, but permits the information to be stored in your application instead of an external file, and it permits direct control of the opening and closing of individual files instead of complete sets of files.

isam1.c is an example of how to use incremental ISAM structures. Other than it's use of incremental ISAM structures it is identical to *isam.c*.

Incremental ISAM structure

This first difference is that you must add the *"ctifil.h"* include file. This contains the incremental ISAM structure typedefs. These will be used to create a series of related structures that will duplicate the information found in the ISAM parameter file. Details on the composition of these structures can be found in the chapter on ISAM functions. We will briefly discuss them in relation to our sample program.

IFIL structure

Each data file will have one IFIL structure. Here we show the structure for the inventory data file. Note that the values shown match those found in the data description record in the ISAM parameter file *isam.p*. When using incremental ISAM structures the file extent ".dat" is added to the data file name automatically. Note that one of the values is a pointer to the IIDX structure that contains information about the indices.

```
IFIL inv_dat = {
   "invent",          /* data file name ("dat" is always
assumed)*/
   0,                 /* data file number      */
   128,               /* data record length    */
   4096,              /* data extension size   */
   1,                 /* data file mode        */
   1,                 /* number of indices     */
   4096,              /* index extension size  */
   1,                 /* index file mode       */
   &inv_idx,           /* pointer to index array*/
   "",                /* r-tree info (not used here)*/
```

```
    ""                  /* r-tree info (not used here)*/
};
```

IIDX structure

The IIDX structure describes the indices for the data file. As before, you should be able to relate the values here with the values of the index description record. The file extent ".idx" is added to the index file name. Note the pointer to the ISEG structure, which contains information about the structure of the keys.

```
IIDX inv_idx = {
    25,                 /* key length                */
    0,                  /* key type                  */
    0,                  /* dup off                   */
    0,                  /* null off                  */
    32,                 /* empty char                */
    1,                  /* number of key segments*/
    &inv_seg,    /* pointer to segment array*/
    "invent"            /* index name ("idx" is always
assumed)*/
};
```

ISEG structure

Each key segment is described by an ISEG structure. If there are multiple segments then the ISEG structure must be an array. There must be an ISEG array element for each key segment as specified in the IIDX structure. It is easy to relate the key segment description record of *isam.p* to the following ISEG structure:

```
ISEG inv_seg = {
   2,25,2
};
```

Initialize the system

InitISAM

When using incremental ISAM structures the procedure for initializing the system is similar to the low level functions. A call to **InitISAM** performs the same function as a call to **InitCTree**, or the initialization record of the ISAM parameter file.

CreateIFile
OpenIFile

Each data file is created or initialized by a call to **CreateIFile** or **OpenIFile**. The address of the IFIL structure for that data file is passed to the function. This call also takes care of opening all associated index files.

All other operations in our example are identical to the ISAM parameter file example *isam.c*. The only differences are in the way you initialize the system and create/open the files. There are two other incremental ISAM functions, which we don't use in these examples. **CloseIFile** can be used to close individual data files and their associated indices, and **RebuildIFile** is used to rebuild a corrupted data file and it's indices.

Please note that the regular **CloseISAM** function will close all open ISAM files, whether they were opened with an ISAM parameter file or as an incremental file.

File Searches and Sets

isam2.c takes the sample that we started with in *isam.c* and adds some functionality. The two features that we add are the ability to list the contents of a file, and Sets.

Listing a file

We have added the function *datalist()* to sequentially list the records in key order. The first step is to ask if the user wants to enter a string to define a Set. This will be discussed in the next section of this chapter. For now let us assume that the user just presses the Enter/Return key, which makes the program call the *doall()* function.

FirstRecord

The process that we want to perform is to find the first key in the index, display the record that it represents, and then find each successive key in turn. To start the process we use the **FirstRecord** function. It will find the first key in the **index** specified. If instead we pass the **data** file number as the parameter, then it will find the record that is physically the first in the file. **FirstRecord** reads the record associated with the key into the buffer specified.

NextRecord

After we have displayed the record we are ready to locate the next record in key sequence. We use the **NextRecord** function for this, again passing the number of the index to use. The **NextRecord** function finds the next record in key order, in relation to the last record found. We continue in a loop until **NextRecord** returns an error value of INOT_ERR, which signifies that we have reached the end of the file.

Sets

A Set is a group of records that have keys that match a certain target. For example, an index contains the following keys:

```
CASE
DISKCASE
DISKDRIVE
DISKETTE
KEY
KEYBOARD
KEYCAP
```

We could define a Set by the target "DISK". This set would contain the values:

```
DISKCASE
DISKDRIVE
DISKETTE
```

In the function *datalist()* we ask the user for a string to define a Set. If the user enters nothing then we list the entire file, as described above. If a string is entered we use this to define a Set and call the function *doset()*.

This function is nearly identical to *doall()*. We will find the first key in the index, display the record, and then go through the index sequentially. The difference here is that we will start with the first key that begins with the Set string, and end with the last one that starts with the Set string.

FirstInSet

Instead of starting with **FirstRecord** we will used **FirstInSet**. It performs the same function, but **FirstInSet** asks for a key to start with, and the number of characters that should be matched. In our example we take the input value and pass it through the **TransformKey** function to make it a perfectly formed key. The length parameter is the length of the input target key. For example, if we had the keys that we show above, we could enter the target key of "DISK", with a length of 4. **FirstInSet** would find "DISKCASE" as the first key of the set.

NextInSet

To go on to the next key we use **NextInSet** instead of **NextRecord**. It performs the same function, but it will stop when the last key that matches the Set is found. For example, it will tell us that we have reached the end of the file (INOT_ERR) after "DISKETTE".

Note that we do not pass the Set target in the **NextInSet** function. This information is stored by c-tree Plus. It also means that you can only process one Set at a time. You can get around this by allocating more memory for additional Sets with the **AllocateSet** function, and then switching from one set to another with the **ChangeSet** function.

Scanning, Updating, Deleting

Continuing to build upon our sample program, we take *isam2.c* and add the ability to scan the index looking for a particular file, and then to either update or delete it. The new program is *isam3.c*. We are going to replace the *datadel()* function with *update()*.

Please note that this program still uses the same data, index, and parameter files as do all the other isam examples.

Start the scan

In the previous examples, if we wanted to delete a record we had to enter an exact match of the key. Sometimes it is desirable to just enter a target that is close to what you want, and to then search up or down from that point. Also, you may not know

exactly what keys are in the index, so you can enter something that is close and then scan forward or backward until you find the one that you want.

GetGTERecord

If we use **GetRecord** as we did before, we can only find a key that exists in the index, as **GetRecord** finds only exact matches. Instead we will use **GetGTRecord**. This will find a key that is equal to the target, but if there is no exact match it will find the next key that would come after the target. Note that we again use **TransformKey** to format the key properly. If we get a return value of INOT_ERR here then we have entered a target that is past all of the keys in the index.

Once we have found a starting point we display the data values that **GetGTRecord** has read into the buffer. The user is given the option of updating, deleting, or scanning up or down through the index.

Scanning the file

NextRecord
PreviousRecord

As we saw when listing the file sequentially, **NextRecord** can be used to find the next record after the one previously found. We can continue to scan forward through the file with **NextRecord**, or we can scan backwards through the file with **PreviousRecord**. This function works identically to **NextRecord** except that it goes the opposite direction through the index.

Update the record.

Once we have found the record that we want to update, we can easily change the values and write the record back to the file. We have changed the *getfld()* function so that it displays the existing value before asking you for the new value. If you just press the Enter/Return key you will get the same value as before.

ReWriteRecord

The **ReWriteRecord** function is used to rewrite the current ISAM record back to the disk. All that you have to do is to pass it the number of the data file and a pointer to the buffer where the updated record exists.

Note that in our example, it is possible to change the value of the "item" field, which is our key field. **ReWriteRecord** will notice that the new record has a different value for that key field than the original record. It will then delete the old key from the index and add the new key automatically. This may not be the best practice in a real application, but it serves to illustrate how **ReWriteRecord** works. If we had multiple indices based on different fields **ReWriteRecord** would make the same change for all key values.

Delete a record

Once we have found the record that we want to delete by scanning to it we can easily delete it by using the **DeleteRecord** function as before. Keep in mind that this deletes the Current ISAM Record.

Advanced c-tree Sample Programs

In addition to this series of sample programs, we also provide a set of more sophisticated ISAM applications. These introduce some more complex concepts, including multiple indices and compound keys. The programs include:

- **ctexmc.c** - this is a very simple program that is used to create the ISAM files for the next two applications.
- **ctexmg.c** - add, update, and scan **fixed length** data files. There are multiple indices that can be used to scan. This sample uses the file *ctexam.p* as the ISAM parameter file.
- **ctvxmg.c** - this is essentially the same application as *ctexmg.c*, except that it works with **variable length** data files. The file *ctvxam.p* is the ISAM parameter file.
- **ctixmg.c** - this is the same as *ctvxmg.c* except that it uses **Incremental ISAM** structures instead of an ISAM parameter file, and it includes **automatic rebuilding**.

ctexmc.c

This is a very simple program that is used to create the ISAM files for the *ctexmg.c* and *ctvxmg.c*. You will be asked for the name of the ISAM parameter file to use: enter "ctexam.p" if you want to create the files for the fixed length example *ctexmg.c*, and "ctvxam.p" if you want to create the files for the variable length example file *ctvxmg.c*.

ctexmg.c

This program is used to add, update, and scan **fixed length** data files. There are multiple indices that can be used to scan. This sample uses the file *ctexam.p* as the ISAM parameter file.

Note that we are building multiple indices, and that one index is a "member" of another index file. When you scan the file you are allowed to choose which index you are going to scan, or you can scan in the physical data record order.

ctvxmg.c

This is essentially the same application as *ctexmg.c*, except that it works with **variable length** data files. The file *ctvxam.p* is the ISAM parameter file.

ctixmg.c

This is the same as *ctvxmg.c* except that it uses **Incremental ISAM** structures instead of an ISAM parameter file, and it includes **automatic rebuilding.**

Additional Samples

As we continue to develop sample programs to explain various details of the c-tree Plus product we will add them to the distribution disks. There will be a comment block at the beginning of each one to explain the concept that is being illustrated.

8. Advanced c-tree Concepts

In this chapter we will highlight a number of advanced c-tree concepts. All of these features are available in all four c-tree modes.

Resources

There are times that you may want to attach auxiliary information to a particular data file that is not easy to integrate into the structure of that file. This could be information such as a file version number, special flags relating to the status of the file, and so forth. Generally this is information that is not repeated in each record of the file.

You could create a special record, with a special key value, that you do not process as you do your regular records. However, this forces you to handle this record (and it's key) as an exception to the regular records, which places a burden upon the programmer.

c-tree Plus provides a special feature to handle this sort of information, called a **RESOURCE**. Resources are special variable length records that are stored in your data file (whether you use fixed or variable length records) that can be accessed via a special set of functions. These records do not require a key in an index, therefore they will not be accessed by your program if you are working with an index or using the ISAM functions.

Resources and Files

There are three general types of resources that can be attached to a file:

- User defined resources; information that you wish to store in the file, such as a file version number, or a counter of some sort. Generally you will use resources to store information associated with a file that varies from the type of information that you will be storing repetitively in the data records. These types of resources can only be attached to **data** files.
- FairCom defined resources; there is a variety of information that, under certain circumstances, FairCom wishes to store in the file. This can be information relating to IFIL structures, alternate collating sequences, special characters for key padding, and so forth. Usually you will not access this information directly, rather it will be available to a variety of c-tree Plus functions that can use it. These resources can apply to both **data** and **index** files.
- Third party resources; as other developers create utilities that integrate with c-tree Plus, FairCom will assign resource identifiers when necessary.

Resources are a feature that can be turned off when you create a data or index file. Generally you will **not** want to do this, as a number of c-tree Plus features will not be available to you for this file.

Resource Identification

Within a given data file, a Resource is identified by three elements:

- Resource Type (unsigned long integer),
- Resource Number (unsigned long integer),
- Resource Name (null terminated string).

Within each data file you can identify a Resource by its unique combination of Type and Number, or by its Name. Note, however, that the Resource Name is not guaranteed to be unique.

Resource Type

Resource Types in the range of 1 to 127 are reserved for use by FairCom. These will be special Resources used by the FairCom Server and d-tree. Resource Types in the range of 128 to 65536 are also reserved. These will be assigned to third party developers by FairCom. Resource Types from 65537 and above are available to the programmer to be assigned as desired.

Generally you will use the Resource Type to gather your Resources into related groups.

Resource Number

There are no restrictions on assigning Resource Numbers. They are used to separate various Resources within a given Resource Type. When you are adding a Resource to a file you can have c-tree assign the next available value for this Resource Type in the referenced file.

Resource Name

You can use the Resource Name as a way to access Resources in a file instead of using the Type and Number. You may find it easier to understand your code if you use a symbolic name rather than a pair of numbers. If you are adding a number of Resources to a data file over a period of time you may not know what Resource Number is being used, particularly since you can have c-tree assign the next available Resource Number. Access via Resource Name can be simpler than trying to keep track of what Number your Resource has been assigned.

On the other hand, you must be careful in assigning Resource Names. c-tree cannot guarantee that each Resource Name is unique within a file. It is possible for you to add a Resource to a file that has a Name that has already been used in this file.

Resource Names are optional. You do not have to assign one to a Resource. The Resource Name is a null terminated character string of any length, although we

recommend that you do not make them too long. Names starting with "FC!" are reserved for use by FairCom. Names starting with "RD!" are reserved for Resources assigned to third party developers by FairCom.

Disabling Resources

Resources are enabled unless you specify otherwise when the file is created. The only situation where you may want to disable Resources is if you do extensive processing of a data file in **reverse physical order**. With Resources enabled you can no longer use **LastRecord** or **PreviousRecord** to traverse the file in **physical reverse order**. You can still use these functions to process the file in **index** order, though. Resources introduce special variable length records into the data file, so reverse traversing by physical position no longer is possible, just as with variable record length files.

To disable Resources for a data file you must use the DISABLERES (disable resource) file mode when the file is **created**.

It is possible to enable Resources for a file at a later time with the **EnableResource** function, as described later.

Resource Functions

Four functions are provided to manage Resources (see the Function Reference chapter for declarations);

- **AddResource**: Add Resource to file,
- **UpdateResource**: Update Resource,
- **DeleteResource**: Delete Resource from file,
- **GetResource**: Retrieve Resource from file.
- **EnableResource**: Enable Resources for a file.

Each of these functions will have a parameter that is a pointer to a Resource Data Block. This is an area that you create that contains the information that describes the Resource, as follows:

Byte Offset	Data Type	
0 - 3	unsigned long integer	Resource Type
4 - 7	unsigned long integer	Resource Number
8 - m	null terminated character string	Resource Name
m + 1 - n	any collection of data types desired	Resource Information

The Resource Information can be any collection of data fields that you desire. You are not limited to a null terminated character string.

AddResource

This function is used to add a new Resource to a file. The combination of Resource Type and Resource Number must be unique to this file, or an error will be returned. c-tree does not, however, check to see if the Resource Name is in use yet. As mentioned earlier, the Name is optional.

If you pass a value of 0xffffffffL as the Resource Number then c-tree will assign the next available Number for this Resource Type in this file. The assigned Number will be placed in the appropriate field in the Resource Data Block.

When you add a Resource to a file a special variable length record is written to the file, containing the information from the Resource Data Block. This is done even if the data file uses fixed length records. Now the information is available to any user accessing the file.

DeleteResource

Use this function to delete a resource from a given file. Since Resource Names are not guaranteed to be unique you must have the proper Resource Number and Type to be able to delete a Resource.

If the Resource record was locked via a **UpdateResource** call (see below) then the lock is released.

Update-Resource

UpdateResource is used to update the information in a Resource. The Resource Type and Number is used to select a Resource, and then the new Resource Data Block is placed in the file, replacing the prior information

NOTE: **UpdateResource** automatically tries for a low level record lock on the Resource record (unless the record is already locked via a **GetResource** call). If the record cannot be locked then an error value is returned. The lock is automatically released when the **UpdateResource** call is finished.

GetResource

This function is used to retrieve a Resource. The value returned by the function is the record address of the Resource record. The Resource Data Block will be written to a buffer that you specify.

When you use **GetResource** to retrieve a Resource you will use one of the following resource mode (or *resmode*) values:

- **RES_TYPNUM**: find the Resource with the matching Type and Number,
- **RES_TYPE**: find the Resource with the matching Type and lowest available Number,
- **RES_FIRST**: find the Resource with the lowest available Type and Number,
- **RES_NAME**: find the first Resource with a matching Name (remember that c-tree cannot guarantee unique Resource Names),

- **RES_POS**: read the Resource at a given record address (which must must have been obtained in a prior call to **UpdateResource**).

In addition you can OR the *resmode* with the value RES_LOCK. This will make **GetResource** ask for a low level exclusive lock on the matching Resource record. This lock is released if you make a call to **UpdateResource** or **DeleteResource** for the same Resource, or with a call to **LockData**. It is important to note that Resource locks are **NOT** released by explicit or implicit calls to **LockISAM**.

Enable-Resource

This function can be used to enable Resources in a file that was created with Resources disabled. You must first open the file with an EXCLUSIVE file lock mode, and then call **EnableResource**.

Resource Example

The following sample code adds a Resource, and then retrieves it.

```
#include "ctstdr.h"
#include "ctdecl.h"

structure {                          /* Resource Data Block structure */
   ULONG  resource_type;
   ULONG  resource_number;
   TEXT   variable_info[1016]; /* This will hold the Resource Name
                                        and Data*/
} my_resource;

my_resource.resource_type    = 0x10001L;     /* 65537 */
my_resource.resource_number = 0xffffffffL; /* c-tree to assign #/*
strcpy(my_resource.variable_info,"M&!resource");
strcat(my_resource.variable_info,"Actual Resource Data");

if (AddResource(datno,&my_resource,
   (VRLEN)(8 + strlen(my_resource.variable_info))) == 0 )
{
        printf("\nThe resource has been assigned number %ld",
             my_resource.resource_number);
        printf("\nThe resource is located at byte offset %ld.",
             GetResource(datno,"MY!resource",&my_resource,1024L);
} else
        printf("\nCould not add resource. Error #%d",uerr_cod);
```

FairCom Defined Resources

There are many resources that are defined by FairCom that could be added to your data and index files. Generally you will not refer directly to these resources - they will be used by other c-tree functions.

Note that if you disable resources for a particular data or index file, the following features cannot be used with that file.

Alternate Collating Sequence

As described on page 5-19, you can specify an alternate collating sequence for an index. This information is stored as a resource in the index file.

File Security

The c-tree file security system depends on resources. If you have disabled resources for a given data or index file, you will not be able to use the security system with that file.

Padding and Delimiter Values

If you change the padding or key delimiter value for an index with the **SetVariable-Bytes** function, the information is stored in the index as a resource.

DODA and Record Schema

DODA (Data Object Definition Array) information, such as is used in FairCom's **d-tree** product, can be stored in a file as a Resource with the **PutDODA** function. **GetDODA** can be used to retrieve this information. For more information refer to page D-4.

IFIL information

Incremental ISAM structure (IFIL) information is stored as a Resource if a file is created with **CreateIFile**. This information is used by **OpenFileWithResource**, **CloseRFile**, and **GetIFile**.

Superfiles

When working on complex applications you may find that you are creating an overwhelming number of data and index files. This proliferation of files can sometimes create problems for the operating environment and the user. Many operating environments place a limit on the number of files that can be open at any given time. c-tree allows you to get around this by using the concept of *virtual files*, but the performance overhead that this creates can be a problem in some cases. In addition, by having a large number of separate data and index files you complicate the user's backup procedure, and increase the risk of getting various files and indices out of synch.

To help manage this problem c-tree Plus offers **Superfiles**. A Superfile is one physical file that can contain a number of logical data and index files. This can be used to limit the number of physical files open, and allows you to move or copy a group of files as a single unit.

Creating/Opening Superfiles and Members

Superfiles are created with the standard **CreateDataFile** function, and opened with the standard **OpenFile** function. You will OR in the value **SUPERFILE** as a part of the file mode parameter.

A superfile create or open will use two consecutive c-tree file numbers. If you create the superfile with a file number of 10, c-tree will use file numbers 10 **and** 11.

Superfile Members

A c-tree file that is physically stored in a superfile is referred to as a superfile member. When you want to create (or open) a member of a superfile, the superfile itself must already be open. The file name for the member must be formed as follows:

```
<name of superfile>!<name of member>
```

The "!" is the name separator. For example, if you have created a superfile with the name "HOST.DAT", and you want to create a member file named "MEMBER.DAT", you would use the name "HOST.DAT!MEMBER.DAT" when you create (or open) the member.

Superfile members are created with the standard low level and/or ISAM level create and open calls, including the **extended** create and open calls.

Closing Superfiles and Members

If you are using low level file closes (**CloseFile**) it is very important that you close all superfile members first, before you close the superfile itself. If you close the superfile before all members are closed then c-tree will close the members of that superfile, but subsequent open operations during the same program execution may report unexpected FUSE_ERR errors.

If you are using ISAM closes (**CloseISAM**) then c-tree will automatically close the members of superfiles before the superfile itself is closed.

Restrictions when using Superfiles

Since we are storing a number of data and index files in a superfile, the concept of processing a file in physical order no longer applies. Records from different files may be intermixed.

If you try to use the **LastRecord** and **PreviousRecord** functions with a data file number, they will return an error code. **FirstRecord** and **NextRecord** will traverse all of the records, but they are not guaranteed to be visited in increasing record offset order.

All of these functions continue to work properly if you use them with an index file number.

Another restrictions is that a superfile cannot contain another superfile.

Batched Operations

To improve performance in those situations in which a group of related records are to be operated on, c-tree Plus offers the **DoBatch** function. Currently this function provides the following capabilities:

- **Retrieve**: all records that have a key that matches the *partial key request* will be loaded into a buffer region. If there are more records than can fit in the buffer, all those that fit will be loaded, and subsequent calls to **DoBatch** can be used to retrieve the rest;
- **Delete**: all records that have a key that matches the *partial key request* will be deleted.

Additional capabilities will be added in the future.

DoBatch is an ISAM level Set function. You will specify an index file to work with, and a partial key value to search with. **DoBatch** will attempt to perform the specified operation on all records with keys that match the partial key value.

DoBatch parameters

The parameters for a **DoBatch** call are:

- *filno*: the index file number to search,
- *request*: a pointer to a *Partial Key Request* structure, as defined later, which defines the set of records to be acted on,
- *bufptr*: a pointer to a buffer region into which the group of related records will be retrieved, if specified,

- *mode*: a value that specifies the type of action to be taken, together with a number of options.

Partial Key Request structure

One of the parameters that is passed to **DoBatch** is the *Partial Key Request* structure, PKEYREQ. This structure is used to define the set to perform the batch operation on, and to return some statistics on the set. The PKEYREQ typedef is defined as:

```
typedef struct pkeyreq {
    LONG  btotal;          /* total entries in set      */
    LONG  bavail;          /* # of entries locked       */
    LONG  breturn;         /* # of objects in buffer    */
    COUNT siglen;          /* significant length        */
    TEXT  target[MAXLEN];/* partial key target         */
    }  PKEYREQ;
```

When you call **DoBatch** you will pass a pointer to a PKEYREQ structure. The only fields that must be initialized are *target*, which will contain the portion of a properly formed key that you want to match with, and *siglen*, which specifies the number of bytes of the key that is to be considered significant. All keys in the appropriate index that match the first *siglen* bytes of *target* will be processed.

After the **DoBatch** function has been called you can check the values in the PKEYREQ structure for the results. *btotal* will tell you how many records match the partial key target. If you ask for locks on the records, *bavail* will tell you how many of the records were locked successfully. *breturn* specifies the number of records that fit in the buffer region.

It may be necessary to make a series of **DoBatch** calls to process a batch. The first call will specify the batch, and return information on the number of records in the batch. If you are reading the records in the batch into the buffer area, and the buffer is not large enough to hold all of the records, then you can make additional calls to process the rest of the records. Finally, you should always make a last call to close or cancel the batch, in preparation for the next call. Only one batch can be active at a time.

First Call

On the first call to **DoBatch** the mode parameter must be one of the following:

- **BAT_GET**: retrieve matching records,
- **BAT_DEL**: delete matching records.

In addition, you can OR in one of the constants from any of the following optional groups:

Verify Option

BAT_VERIFY Verify that the keys and records match. If a record is selected based on its key value in the index, the data record will be read and the key recreated from the information in the record, to double check that the key is accurate. This prevents a record from being selected if it is in the process of being updated, and the index value has not yet been rewritten.

default The default is to have no verification.

Result Option

BAT_RET_REC Return the actual records to the buffer. As many records will be placed in the buffer region as can fit, and the number that are actually read will be placed in the *breturn* value of the PKEYREQ structure.

Fixed length records will be stored with the 4 byte record offset first, followed by the actual data record itself. Variable length records will be stored with the 4 byte record offset first, then the 4 byte record length, followed by the actual data record itself. It is up to the programmer to use the buffer correctly.

BAT_RET_POS With this option only the 4 byte record offsets of the matching records will be placed in the buffer. This allows you to do the actual reading of the records when you want to.

default The default is for the buffer to not be used. Only the first three fields of the PKEYREQ structure (*btotal, bavail,* and *breturn*) are updated.

Lock Option

Note that since this is an ISAM function, all locks can be released with a **LockISAM(FREE)** function call.

BAT_LOK_RED With this option a read lock will be placed on each record. This lock prevents other users from acquiring a write (update) lock on this record. If a lock cannot be obtained on a particular record then it will not be read. The PKEYREQ value *bavail* specifies the number of records that have been locked successfully. If *bavail* is less than *btotal*, then not all records were locked.

BAT_LOK_WRT Similar to BAT_LOK_RED, except that a write (update) lock will be acquired.

default No locks will be acquired.

Completion option

BAT_COMPLETE When BAT_COMPLETE is specified **DoBatch** will return a successful completion value only if ALL matching records are successfully locked. If a lock cannot be obtained on any record, **DoBatch** will signal failure, and all of the locks will be released. This option makes sense only if you specify BAT_LOK_RED or BAT_LOK_WRT.

default Ignore unlocked records, allow successful completion even if a lock has not been obtained on all records.

Continuation Calls

The *breturn* value of PKEYREQ will tell you how many records were read into the buffer. If *breturn* is less than *bavail* then the buffer was not large enough to hold all of the matching records. You can make another call to **DoBatch** to retrieve the rest of the records.

To process the next group of records in the batch you will call **DoBatch** with a *mode* value of **BAT_NXT**. *btotal* and *bavail* are not changed, but *breturn* reflects the number of records in the buffer for this **DoBatch** call.

Complete or Cancel Call

When you are finished with a batch you should make a call to **DoBatch** with a mode of **BAT_CAN**. You can use this to cancel a batch before you have processed all records. It is also a good idea to make this call even if you have processed all of the records. There are some cases where **DoBatch** will close the batch automatically. However, this doesn't always happen. For instance, if you do not specify either BAT_REC_RET or BAT_RET_POS, so that the only effect is for the PKEYREQ structure to be updated, **DoBatch** does NOT automatically close the batch. If you try to make another call of that same sort you will see error BTST_ERR.

Therefore, we recommend that you always make a **DoBatch** call with a *mode* of BAT_CAN.

Non-Server Multi User

c-tree Plus supports traditional **multi user file access**, where multiple programs or users can have *direct* access to the database simultaneously, with file and record locking under control of the application programmer. Each program accessing the database will maintain it's own buffer area, and will read from and write to the files and indices directly. A File Server is *not* used.

This section discusses how to write a program using c-tree Plus in traditional multi user mode. The basics of multi user access have already been discussed in Chapter 6. If you are going to use the FairCom Server or the FairCom SQL Server, you do not have to read this section.

9. Non Server Multi User

There are significant differences in some aspects of c-tree Plus when comparing traditional multi user mode (not using a Server) with Server mode (using FairCom Server or FairCom SQL Server). This chapter discusses the specifics of multi user mode when **not** using a server.

With traditional multi user mode the operating system has a great deal of influence on c-tree Plus. As discussed in Chapter 6, we use both write locks and read locks. However, not all operating systems supply both. In addition, the manner in which c-tree Plus uses write locks may vary from the way that the operating system implements them.

Two types of record locks must be distinguished in c-tree Plus: record locks applied to the B-Tree nodes in the index file, and record locks applied to a program's data records. While the reason for the locks is almost identical (viz., to avoid simultaneous and conflicting updates to the data), there are significant differences in the management of these two classes of locks. c-tree Plus automatically takes care of any required node locks, while your application program must control the data record locks.

Node Locks

A B-Tree node corresponds to a record in a c-tree Plus index files.

In the non-server, multi-user mode of operation, when two programs attempt to update the same node, one must wait until the other has completed its update. The c-tree Plus algorithm has all the necessary calls to the locking routines coded into place.

Because of the robust nature of the c-tree Plus B-Tree algorithm, it is not necessary for a program that is merely searching (i.e., not updating) the index file to be locked out of any node, even if the node is currently being updated by another program. This leads to a special case: if only one "user" in a non-server, multi-user system is allowed to update the index files, then it is not necessary to actually place locks on nodes which are being updated.

Another important aspect of the special B-Tree algorithm used in c-tree Plus is that competing node locks are guaranteed not to cause a deadlock situation. Therefore, the node lock routines can ignore the problems associated with deadlocks.

In order to use the c-tree Plus index files in a non-server, multi-user system, the following steps must be taken:

- set the disk protocol in *ctoptn.h* to FPUTFGET; and

- if more than one user is allowed to update the index structures at the same time, the node lock routines (LOCK and UNLOCK) in *ctclib.c* must be coded to access the operating systems locking protocols.

For most environments, these routines have already been implemented. If your operating system/compiler environment is one for which c-tree Plus LOCK and UNLOCK have not been implemented, and if you intend to run in the multi-user non-server mode of operation, then examine the compiler's run-time library documentation to determine how locks are implemented. Also, examine the implementation of locks in the *ctclib.c* modules supplied for the Microsoft C compiler under PC/MS-DOS and the Unix system, and the *ctclib.d* module (as these will serve as examples for LOCK and UNLOCK).

NOTE: the most significant feature of LOCK is that it loops until it acquires a node lock. The looping is permitted since the index routines are guaranteed deadlock free.

Data Record Lock Implementations

For non-server, multi-user implementations, the same basic steps are required for the data record DLOCK, RLOCK and UDLOCK routines as are required for the node LOCK and UNLOCK routines described above. If your system is not already supported by an appropriate *ctclib.c*, then keep in mind that the data record locks are NOT guaranteed to be deadlock free. Therefore, where the (node) LOCK routine can wait for a lock (since they are guaranteed deadlock free), the DLOCK and RLOCK routines must return immediately if a lock conflict is encountered.

Error code 42 (DLOK_ERR) signifies that another process owns a conflicting lock on the same data record.

Dummy Lock File

The non-server, multi-user mode of operation uses a dummy lock file in those environments which do not support write locks. For example, DOS 3.1 only supports exclusive record locks: no other process can read or write an exclusively locked record. c-tree Plus does not require such strong locks. A write lock allows other processes to read the locked record, but not write it. To simulate write locks, c-tree Plus secures true exclusive record locks on phantom regions of the dummy file.

If two applications share one or more files in a multi-user c-tree Plus environment which uses a dummy file, then the following conditions must be met:

- they must use the same dummy file (which must be the zeroth file in the ISAM parameter file or be assigned file number 0 if using low level functions);

- files shared between the applications must be assigned the same c-tree Plus file numbers; and
- files **not** in common must **not** be assigned the same file numbers.

In order to make it easier to assign file numbers consistent with the above three points, remember that you can inflate the *idxs* parameter of the parameter file or the *fils* parameter in **InitISAM** to ensure sufficient flexibility in numbering the files. For example, if you call **InitISAM** with *fils* set to ten (10), but you are only using two files, they can be assigned any file numbers between zero (0) and nine (9).

The example below is the same sample ISAM parameter file that is found on page 5-20, modified to add the dummy lock file.

```
18 3 4 2

0     cust.lok    128     0  3  0
4     cust.dat    128  4096  1  3
 1    custnam.idx  24 0 0  1  4096  1 1  32  3
    6  14 2
   30   6 2
  121   4 3
 2                   4 0 0                0   0  1
    2   4 1
 3    custzip.idx  22 0 0  0  4096  1 1  32  3
  112   9 2
    6   9 2
  121   4 3
```

The dummy lock file is assigned data number 0. This file is optional file to enable write locks on non-server multi user systems which only support exclusive locks.

NOTE: when used, a dummy lock file must always be opened as a permanent file, *NOT* a virtual file. Use a file mode of 3.

Instead of ensuring that common files receive the same c-tree Plus file numbers, you may wish to use your own unique, small integer as the basis of identifying the file in the lock routine. Ordinarily c-tree Plus left-shifts the c-tree Plus file number as a means of partitioning the dummy file among the "real" c-tree Plus files. (See the *ctclib.c* modules for DOS 3.1 for examples of this left-shifting operation.) If you use your own unique integer as the basis of the left shift operation in the locking routines, then the c-tree Plus file numbers need not be common across different applications. By unique, we mean that each file is assigned its own number, and this number is used no matter which application uses the file.

With the existing code, the node offsets to be locked are shifted as follows in LOCK and UNLOCK (both found in a *ctclib.c* module):

```
node |= ((LONG) knum->filnum << LEFT_SHIFT);
```

where *knum-filnum* is the "keyno" value from your parameter file. Assuming you have created a function called unique which maps a c-tree Plus file number to your own unique value, then the node offset would be treated as follows:

```
node |= ((LONG) unique(knum->filnum) << LEFT_SHIFT);
```

The same type of modifications would be made to DLOCK and UDLOCK for the data record offsets.

Dummy File Alternative

Instead of locking a corresponding region of a dummy file for each lock on a "real" file, one can lock a "dummy" region of the real file which is uniquely related to the actual region. Again, our intent is not to lock the actual region, but to lock a uniquely related region that never has to be read. This can be accomplished by adding a suitably large offset (e.g., 64MB) to the actual byte position of the lock. In this way, the lock is placed on the actual file involved, and no file numbering convention is required. The size of the offset must clearly exceed the largest possible actual record offset for your application.

However, you must be cautioned that the c-tree Plus file size extension procedure closes and reopens a file on some systems (e.g., PC/MS-DOS) when the *xtdsiz* parameter is non-zero. The file closing will cause errors if locks were pending on the file. The dummy file approach is not subject to this problem since the locks are not actually on the file which is closed and reopened during file size extension. File size extension is not a problem under UNIX and XENIX since we use the *sync()* function to force directory updates to disk instead of closing and reopening the files.

Remembering that this technique cannot be used with a non-zero *xtdsiz* parameter if file size extension causes the file to be closed and opened, the following code segments show how to modify the standard locking approach. As before, the code in LOCK is as follows:

```
if (knum->clstyp != VAT_CLOSE)
  node /= knum->recsiz;
node |= ((LONG) knum->filnum << LEFT_SHIFT);
if (ctseek(ct_key,node))
  return(uerr(SEEK_ERR));
while (locking(ct_key->fd,LK_NBLCK,1L))
```

The above code would be changed to:

```
node |= 0x4000000L; /* add 64Mb offset */
if (ctseek(knum,node))
  return(uerr(SEEK_ERR));
while (locking(knum->fd,LK_NBLCK,(LONG) knum->recsiz))
```

Similar changes would be made to UNLOCK, DLOCK, RLOCK and UDLOCK. However, be careful if you modify the existing *ctclib.c* modules for the DOS environment: they do not keep track of the *lokcnt* header member (because the dummy file eliminates the need for this information). If you use this 64MB offset technique with a DOS locking routines, be sure to include increments and decrements for *lokcnt* upon each lock and unlock operation, respectively. See *ctclib.c* for UNIX for examples of how to perform the *lokcnt* updates.

FairCom Server

The FairCom Server provides a flexible, high performance implementation of the c-tree Plus index and data file functions for multi-tasking and network systems.

In addition, there are a number of advanced capabilities that are available only if you are using the FairCom Server.

If you are going to use the FairCom Server you will need to study the material in this section after you have mastered the Basic c-tree Plus concepts.

10. File Server

The FairCom Server provides a flexible, high performance implementation of the c-tree Plus index and data file functions for multi-tasking and network systems. Because of the clean separation between the server and the implementation dependent message handling (which is required to send messages between an application and the server), the FairCom Server is implemented in different environments by changing only the message routines.

The main advantages to using the FairCom Server instead of a non-server mode are:

- **Reduced disk I/O:** in the non-server mode, applications cannot perform local file buffering, or else the applications would have inconsistent views of the files. Therefore, all the disk I/O is forced to and from the disk. In the server mode, the server performs all the disk I/O, and it can perform as much buffering as you desire.
- **Reduce traffic on Networks:** in network systems the server reduces the amount of message traffic sent over the network (provided the node running the FairCom Server is also the node where the c-tree files are stored.) In the non-server mode, all the disk I/O must be sent over the network. With the server, only the application request and the server response must be routed over the network; the intermediate results are not sent over the network.
- **Transaction Processing:** a sophisticated set of functions are provided to provide you with industrial quality on-line transaction processing, including full commit and rollback, savepoints, and complete logging.
- **File Security:** file passwords, user passwords and ID's, groups, and other sophisticated features to provide flexible and powerful file security.

Server-Application Relationship

The Server executes all of the c-tree Plus functions. When you link your application to a c-tree application library set up for the server, your application becomes bound with the required Server system dependent message handling functions. The resulting executable module can only run on a system designed to support the message handling routines included in the link step, and on which the FairCom Server is running. This executable module cannot run as a stand-alone, single-user application.

NOTE: to run an application linked for the server in stand-alone, single-user mode, it is not necessary to change the application code. It is necessary to relink the application with a single-user c-tree Plus library.

Logging On and Off the Server

Whenever you are using the FairCom Server your application must establish communications with the Server. This is called *logging on*. If you are using ISAM routines you are automatically logged on to the Server when you make a call to **InitISAM**, **OpenISAM**, or **CreateISAM**. If you are using low level functions, the call to **InitCTree** logs you on to the Server.

In addition, when your application is finished, you need to tell the Server that you no longer need its attention. This allows the Server to reallocate the memory used for this application to other uses. If you are using ISAM routines the **CloseISAM** function will log you off of the Server. If you are using low level functions, you must use the StopUser function to log off.

File Modes

The FairCom Server uses all of the file modes as described on page 3-28 and page 6-1. In addition, there are file modes for record locking, managing transaction processing and I/O management.

Record Locking mode

Although these modes are specified when you open or create a file, they affect how the server handles record locks.

CHECKLOCK If you add the file mode CHECKLOCK when you open or create a file, c-tree will require that you have a **record lock** before a record can be updated. If you do not, the error code DADV_ERR will be returned.

CHECKREAD This is similar to CHECKLOCK except that it applies to record **reads** instead of **updates**. You must have at least a read lock on a record before it can be read, otherwise the function will return error code DADV_ERR.

Transaction mode

PREIMG
TRNLOG WIth the FairCom Server you can specify the method of transaction management for each file. Transaction management is discussed in more detail in Chapter 11.. TRNLOG is used to specify both automatic recovery and transaction atomicity (a term we will define later). PREIMG is more restricted, providing only transaction atomicity. If neither is specified, then no transaction management will be done for this file. If you are going to use transaction management then every file that must be kept synchronous with other files should have one or the other mode specified.

I/O Management

DUPCHANEL

In certain multi user environments it is possible to allow c-tree Plus to use more than one file handle or I/O channel for a given file or index. If the operating environment supports this it can significantly improve the file access performance of c-tree Plus. If, for instance, you have two users reading from and writing to the same file at the same time the FairCom Server can improve the performance of the system by using multiple file handlers. Once you have established this by using the DUPCHANEL file mode the operation is transparent to you - the FairCom Server manages it automatically. This mode should be specified for files that will have heavy use by multiple users simultaneously. The benefits may be decreased if you have a large number of files open that have been created with this mode. In addition, if you have a large number of files and a restricted number of file handles available from the operating environment, you may not see an improvement.

Transaction Processing

There are two major aspects to transaction processing, **atomicity** and **automatic recovery**. These are related yet different aspects to transaction processing, and not all products supply both. The FairCom Server provides a set of functions and file modes that cover both aspects of transaction processing.

Atomicity

Often when you are updating a database you will be performing several functions in a group. For instance, when creating an invoice you may be updating several files, the account balance in the customer file, the invoice file, an invoice detail file, inventory records, and others. It is important that **all** of these actions take place to keep the files synchronous. If for some reason some of the actions take place, but not others, your files are out of synch, and it can be difficult to correct the problem later. If one action cannot take place it would be best to not let **any** take place. We call this **atomicity**. c-tree Plus provides a set of functions which provide you with this feature. You can mark a set of operations so that none will take place unless they can all take place. c-tree Plus goes beyond this, allowing you to create "savepoints" where you can partially back out a group of operations, and "roll back" transactions to a given point, so that you can restore your database back to a state that it was in sometime in the past.

Automatic Recovery

Once you have established transaction processing you can take advantage of the **automatic recovery** feature of the c-tree Plus file server. Atomicity will generally prevent problems of files being partially updated. However, there are still some situations where a system crash can cause data to be lost. Once you have signaled the end of a transaction there still is a "window of vulnerability" while the file server is actually committing the transaction updates to disk. In addition, for speed considerations some systems may buffer the data files and indices, so that updates may not be flushed to the disk immediately. If the system crashes, and one of these problems exist, the file server will detect it. If you have set up the system for automatic file recovery then the file server will automatically reset the database back

to the last, most complete state that it can. If any transaction sets have not been completed (or "committed"), they will not have any effect on the database.

Transaction processing is covered in more detail in Chapter 11.

File Numbering Relaxation

On systems that do not support record write locks, and for which a FairCom Server is **NOT** used, c-tree utilizes a dummy file to permit exclusive locks to operate as write locks. However, this requires the same file to be assigned the same file number by different applications, and different files to be assigned different file numbers.

The FairCom Server permits different applications to assign the same file different file numbers, or to assign different files the same file number without causing problems.

11. Transaction Processing

When using the FairCom Server, c-tree Plus offers a complete set of functions and file modes that will provide you with a wide variety of transaction processing capabilities, including **atomicity** and **automatic recovery**, as introduced in Chapter 10.

Why do we need Transaction Processing?

Let's examine the example that we used earlier, entering an invoice in an accounting system. When we create an invoice, several data files may be updated. The customer master file may have a field for each customer that keeps the current balance. This balance must be increased if we issue an invoice. There is an invoice file that has a record added to it. There may be a separate invoice detail file, keeping a record of each line on the invoice. In addition, each invoice detail will affect the balance of an inventory item. All of these files must be in synch - the customer balance must match the total of the open invoices, the detail lines must all be present, and the inventory balances must be accurate as affected by the invoices and other activities.

What happens if the computer crashes while these files are being updated? You could possibly have some of the files already updated, but others not. For instance, the customer and invoice files could be updated, but the inventory records may not be. Possibly, depending on the way the operating environment works, those records that already exist (the customer and inventory records) could be updated, but the records that are to be added to the file (the invoice and invoice details) may not be saved yet. In any case, you now have files that are not synchronous. Adjusting the files to bring them back into balance can be difficult, as you may not be sure which files have been updated and which have not. Often your only choice is to go to a backup copy of the files, where you know that all of the files are synchronous. Unfortunately, these backups could be days out of date (or more!), and it can take time to make them current.

It doesn't have to be a disaster situation to cause problems, either. The file updating in our example requires a number of files and records, and the user has to have the proper file and record locks in order to be able to complete the processing. What if you find that you are part way through the updating process and you cannot get one of the records to update? For instance, if you have a large number of details in the invoice, you may find that part way through the process one of the inventory records is locked by another user. Do you just skip this item? If you have done a large number of updates already, for the other details in the invoice, how do you go back?

The answer to these problems is **transaction processing** with c-tree Plus and the FairCom Server.

What the FairCom Server Provides

atomicity

The basic purpose of transaction processing is to guarantee **atomicity**. This means that if we have a group of operations that we are processing, either all of the group will work or none of it will. The reasoning is that we may have several data files, or possibly several data records in one file, that we want to keep synchronous.

There are two situations that we can consider here. Looking at the example above, we want to be able to back out of a group of actions if we decide that we don't want to complete it (as in the case of not being able to get a necessary lock). In addition, if there is a disaster of some sort (such as a computer system crash) we want to guarantee that the database is restored to it's most complete state. We want all of the file updates in a group of actions, both the modification of existing records and the creation of new ones, to either take place or not, as a group.

Automatic Recovery

Even if we can provide atomicity, there is a chance that some of the information in a committed transaction may not be recorded in the data files and indices if the system crashes at a particular moment. If we are in the middle of a transaction, atomicity guarantees that a transaction in progress will not affect the database. However, when we commit the transaction at the end, there is a "window of vulnerability". While the file server is actually doing the file updates as a part of the commitment of the transaction, which can take some time if the transaction is large, the system could crash. This would leave you with a partially committed transaction. Another potential problem can occur with data buffering. If the file system is heavily buffered, updates which have been committed may still be sitting in a buffer somewhere, not yet written to disk. What is needed is a form of transaction *write ahead logging*, which will log the transactions to the disk (without buffering) so that if a crash occurs when commitment is underway, the information can still be recovered. This recovery should be an automatic feature, so that the database is maintained without user intervention.

The FairCom Server offers the most complete set of transaction processing functions and capabilities of any file management product on the market today. By using these functions properly we can provide both **atomicity** and **automatic recovery**.

Basic Transaction Processing

Many of the benefits of transaction processing can be achieved very easily. This section will introduce you to the basics of establishing transaction processing. For many users, this will be all that is needed.

Detailed explanations of the functions used here will be given later, including additional functions and other options.

Create Files

When you create your data files and indices you must specify if that file is to be involved with transaction processing. To get the full benefits, including atomicity and automatic recovery, then every file the requires recoverability should have the file mode of TRNLOG. Just "OR" it in with the other file mode values.

Begin Transactions

You will need to decide on logical groups of file updates that can be delimited as transactions. Record locks are held on updated records for the duration of the transaction, so you don't want to make the transaction group too large. On the other hand, you do not want to make the transaction group too small. You must contain all of the related updates in one transaction. Using our example from above, you don't want to have the transaction group involve more than one invoice. You also don't want it to involve less than a whole invoice.

To begin a transaction make the following call to the **Begin** function:

```
Begin(TRNLOG | ENABLE);
```

The ENABLE parameter eliminates the need for a call to **LockISAM**.

End Transaction

When you have finished all of your updates for this transaction, you must end it to "commit" the transaction. Until you have done this none of the actual transaction updates will occur. The **Commit** function is used to do this. In addition to committing the transaction, this function can also free all locks created during the transaction.

```
Commit(FREE);
```

That is all you have to do to get complete transaction processing protection in your application using c-tree Plus . There are, however, other variations on how you can use these functions, as well as additional functions.

Note that if an file update error has occured during the transaction, and that error has not been resolved properly, c-tree Plus will not allow you to successfully commit the transaction. This preserves the integrity of your data.

File Modes

When you are creating your data files and indices you have three options in relation to transaction processing. These options are established by using the appropriate value in the file mode for the file that you are creating..

TRNLOG

If you use a file mode of TRNLOG you will get the full benefit of transaction processing, including both atomicity and automatic recovery. If you are not sure of what mode to use, and you do want to use transaction processing, then use this mode.

PREIMG

The PREIMG file mode implements transaction processing for a file, but does **not** support automatic file recovery. Files with PREIMG mode do not take any space in the system transaction logs.

No Transaction Processing

If you do not use either PREIMG or TRNLOG then the file will not be involved in transaction processing in any way. If you are using transaction processing then you should only do this with files that do not need to be kept as a part of your database.

Please note that you should only use one or the other of PREIMG and TRNLOG.

Record Locking

Record locking requirements are the same with and without transaction processing. Records to be updated or deleted should be locked. Use of CHECKLOCK as a file mode will ensure that no updates are performed without locks.

Begin

When you invoke **Begin** you can add the constant ENABLE to the transaction mode. This will automatically invoke a **LockISAM**(ENABLE) call, so that all ISAM update functions will automatically lock records. During the transaction you can modify the ISAM lock mode by further calls to **LockISAM**.

Low level functions will still require that you explicitly lock the records with **LockData**.

Commit

At the end of the transaction you have to options in the **Commit** call;

- You can use a mode of FREE to free all locks, both ISAM and low level locks, automatically (only for those locks initiated during the transaction).
- You can use a mode of KEEP to hold onto the record locks. To free those locks you have to make a **LockISAM**(FREE) call, or a **LockData**(FREE) call for each low level record lock.

Marking Transactions

In your application program you will be marking the beginning and the ending of the transaction with **Begin** and **Commit**, as mentioned above. All file updates that occur during this time will be held by the file server until the transaction completes successfully. When this occurs the file server will actually process all of the file updates.

Begin

You use the **Begin** function to mark the beginning of the transaction. This function will return a **transaction number** that you may want to use in some situations. You will pass a mode to **Begin**, which can be either TRNLOG or PREIMG. In addition, you can include ENABLE with the mode so that ISAM functions will automatically lock records.

When you start a transaction with **Begin** the file server will look to see what files are open at the time, and what their transaction file mode is. The file modes of all open files must be compatible with the mode used in the **Begin** call.

- Files with no transaction mode will be ignored by **Begin**
- If the **Begin** mode is PREIMG, then all files that are open must have been created with a file mode of PREIMG, or no transaction processing at all. If any of the files were created with a file mode of TRNLOG then **Begin** will return the TTYP_ERR error code. You cannot skip transaction logging if the files have been set up for transaction logging.
- If the **Begin** mode is TRNLOG then transaction logging will not take place if a file was created with a file mode of PREIMG.

If you created a file with a file mode of TRNLOG, then you must use TRNLOG with your call to **Begin**.

Commit

The function **Commit** will end the transaction, and **commit** it. This means that all of the updates that have taken place since the last **Begin** call will actually take place. **Commit** requires a parameter of either FREE or KEEP. FREE will free all ISAM locks held for the transaction, KEEP will keep all the locks (they can be released later with a **LockISAM**(FREE) call.

Note that if an file update error has occured during the transaction, and that error has not been resolved properly, c-tree Plus will not allow you to successfully commit the transaction. This preserves the integrity of your data.

Aborting a Transaction

There may be situations where you do **not** want to commit the transaction, rather you will want to throw the whole thing out as if it never took place. For instance, you are taking a customer order over the telephone, updating the files as the customer requests certain items. However, the customer decides that the order should not be placed at all, and cancels out. Without transaction processing you would be faced with the difficult process of going back and reversing out all of the updates that you have done. Alternately, you could build your system so that you don't actually do the updates, you just keep them in a temporary file. This can also be a problem, as you are not able to place a commitment on a particular inventory item, and when you actually decide to place the order you may find that the items you promised were consumed by another order.

Abort

This process is made simple by transaction processing. Start the order with a **Begin** call, log the order details as you go, and issue a **Commit** call to commit the updates when the customer decides to place the order. However, if the customer decides to back out, you can use the **Abort** call to abort the transaction. This will abort all of the file updates since the last call to **Begin**, as well as releasing all locks.

Note: starting a transaction in an interactive mode will cause long held locks on those items updated but not yet committed.

Savepoints

There are times when you may want to abort a portion of a transaction, but not all. You may be processing several optional paths of a program, going down one branch, then backing out and trying another branch. It may be possible that you don't want any of the updates to occur until you are completely finished, but you want the flexibility to back out a part of the updates. Another possibility would be if you have run into some form of update error, such as an **AddRecord** failing due to a duplicate key. You would want to back up to a prior point, correct the problem, and continue. The FairCom Server lets you implement this by using **savepoints**.

SetSavePoint

A savepoint is a temporary spot in the transaction that you may want to back up to without having to abort the entire transaction. During a transaction, when you want to put a place mark in the process, issue a **SetSavePoint** call. This does **not** commit any of the updates. The function returns a **savepoint number**, which you should keep track of. You can make as many **SetSavePoint** calls as you wish during a transaction, and each time you will be given a unique savepoint number.

RestoreSave-Point

When you decide that you want to back up to a savepoint, issue a **RestoreSave-Point** call. You will pass this function the savepoint number that you want to back up to. This returns you to the point at which you issued the specified **SetSavePoint** call, without aborting the entire transaction.

Record locks that are held on actions after that savepoint will be released, unless they were also locked by actions prior to the savepoint.

Errors in Transactions

The primary purpose of Transaction Processing is to protect the integrity of your data. This is particularly important if you are updating multiple files in one transaction. Earlier in this chapter we used the example of an invoicing system, where we are updating a customer file, an invoice master and detail file, and an inventory file. We either want the entire invoice process to complete, or none of it. A partially entered invoice is not acceptable.

If an update error occurs, an error flag is set. If you try to use **Commit** to commit this transaction, an error code will be returned and the transaction will be aborted.

If an update error occurs we typically have two choices; abandon the entire transaction, or go back to a savepoint and try again. The choice is yours, and the correct choice may depend on what error occured and how you can correct it.

Abort

To abandon the entire transaction you will use the **Abort** function. This will throw away all update actions that occured since the initial **Begin** call. You can start another transaction, or take some corrective action.

RestoreSave-Point

If you have been setting savepoints with the **SetSavePoint** function you can back up to an appropriate place in your transaction with the **RestoreSavePoint** function. This allows you to go back part way into the transaction and then continue on, without throwing everything away.

ClearTranError

There is one other option that you have, but it must be used with caution. If an update error has occurred you can use the **ClearTranError** function to clear the error flag and then continue. If you do this you are circumventing the safeguards that transaction processing has set. This will allow you to go ahead and commit the transaction. Generally you should only do this if you are working with low level functions and you know how to correct the problem.

If you use ClearTranError then YOU are responsible for maintaining data integrity!

Automatic Recovery

Once you have established transaction processing you can take advantage of the **automatic recovery** feature of the FairCom Server. If you establish transaction logging, a buffer is kept where the transactions are written to prior to commitment to the database. This buffer is a write thru buffer, so that the information is guaranteed to reach the disk.

If the file server crashes for some reason (anything from a software problem to power failure of the hardware) while the server is committing a transaction, the file server will detect that a problem has occurred then next time it starts up. The server will then automatically reset the database back to the last, most complete state that it can, using the transaction log. If any transaction sets have not been completed (or "committed"), they will not have any effect on the database.

Only those files that were created with a transaction mode of TRNLOG will be affected.

However, if the disk media fails containing your files or the system logs, automatic recovery will not be able to function properly. The FairCom Server provides a dump facility to create backups (even while the Server is operating). See the Server documentation for details.

Automatic recovery also comes into play if the file system is heavily buffered (which dramatically improves performance). With buffering, updates which have been committed may still be sitting in a file buffer somewhere, not yet written to disk. This can be offset by using the file mode of WRITETHRU, but that may slow the system down noticeably. Don't use WRITETHRU with TRNLOG files. WRITETHRU is not necessary because the file server can detect if these buffered transactions have not yet been written to the disk, and will use the transaction log to complete them.

12. Security

Any time that you have multiple users accessing a database you have to be concerned with file security. This is particularly important when you have programs that allow easy access to the database, such as FairCom's **Natural Query™** natural language file inquiry program. You must be able to prevent unauthorized access to restricted files, you may want to limit who can add or change records in the database, and it might be important to control who can delete files.

The FairCom Server supports three complementary forms of system security: user passwords, file permissions, and file passwords. By careful use of these features you can provide complete control over who can access a particular file, and what that user can do with the file once it has been accessed. It is important to note several things at this point: c-tree Plus file security is only available if you are using the FairCom Server (or the FairCom SQL Server); file security is a function of access via the server, therefore access by non c-tree programs may not be limited in many operating environments.

Accessing File Security

Extended Functions

In the first several chapters of this manual we mentioned the **extended functions**. These are functions that perform the same operations as their complimentary **standard functions**, with the additional ability to access the c-tree file security features. These functions have a suffix of **Xtd** (for instance, the extended version of **InitCTree** is **InitCTreeXtd**). The extended functions are:

- **InitISAMXtd**
- **InitCTreeXtd**
- **CreateIFileXtd**
- **OpenIFileXtd**
- **RebuildIFileXtd**
- **CreateDataFileXtd**
- **CreateIndexFileXtd**
- **CreateISAMXtd**
- **OpenISAMXtd**
- **OpenFileXtd**
- **OpenFileWithResource**

Security function

In addition to the extended functions c-tree Plus provides the **Security** function to manipulate file passwords and permissions.

Forms of System Security

Users, Groups, and Passwords

System Administrator

The FairCom Server comes with a System Administrator utility program, as described in the FairCom Server documentation. This utility performs a number of security related tasks.

User ID

When using the FairCom security feature, each user that will use the Server must be given a User ID by the System Administrator. The ID is a 16 byte ASCIIZ string (i.e., a null terminated ASCII string).

User Password

Each User ID can have an optional user password. The password is a 10 byte ASCIIZ string.

Groups

The System Administrator will create one or more Groups. Each group is assigned a unique Group ID which is a 16 byte ASCIIZ string. In order to provide a convenient way to permit users with related needs to share information, each user is assigned membership in from one to sixteen groups. The administrator adds the groups to the system and assigns users to the groups. Each user has a default group (which counts as one of the user's sixteen groups).

When an application program logs on to the server with one of the extended functions (**CreateISAMXtd**, **OpenISAMXtd**, **InitISAMXtd**, or **InitCTreeXtd**) it will send a User ID and user password to the server. If you log on with one of the equivalent standard functions no User ID is sent, and the user is automatically assigned the User ID of GUEST and membership in the GUEST group. If a User ID and/or password are sent which do not exist, the initiating call to the server returns an error code and the program is not connected to the server.

File Permissions

When the server creates a file, the User ID of the creating user is assigned to the file as the file's **owner**. The owner's default group is also assigned to the file unless the creating function specifies one of the owner's other Group ID's.

Further, the user may specify a **permission mask** which determines the kind of access that users may acquire on subsequent opens. The mask is comprised of three components: **owner** permissions, **group** permissions and **world** permissions. With this you are able to allow different users different levels of access to the file.

When the **owner** of a file opens that file, that user is given owner's permissions. Typically this level allows the user the most flexibility in managing the file. If a user who is not an owner of the file opens that file, c-tree will look at the groups that the user belongs to. If that user is a member of the **group** assigned to the file then the user is granted group permissions. Finally, a user not falling into these categories is

granted **world** permissions. Generally, permissions become more restrictive as you go from owner to world.

For example, an owner of a file may be able to read, write, and update records, and delete the file if necessary. Another user who is a member of the group for that file could have permission to read, write, and update records, but not to delete the file. Finally, a user who is not an owner, and not a member of the file's group, may have permission only to read the file.

Permission Mask

When the file is created you will provide a **permission mask** for that file. The permission mask is formed by "OR"ing the appropriate permission constants (found in *ctport.h*) from the following list:

Permission Constant	Description
`OPF_READ`	owner read permission
`OPF_WRITE`	owner write/update permission
`OPF_DEF`	owner file definition permission
`OPF_DELETE`	owner file deletion permission
`OPF_ALL`	owner granted all permissions
`GPF_NONE`	group access denied
`GPF_READ`	group read permission
`GPF_WRITE`	group write/update permission
`GPF_DEF`	group file definition permission
`GPF_DELETE`	group file deletion permission
`WPF_NONE`	world access denied
`WPF_READ`	world read permission
`WPF_WRITE`	world write/update permission
`WPF_DEF`	world file definition permission
`WPF_DELETE`	world file deletion permission

"READ" permission grants access to data, but no ability to change the data. "WRITE" permission grants update abilities. "DEF" permission allows the user to change or set file definition characteristics such as alternative collating sequences and record schemas. "DELETE" permission grants the ability to delete the entire file.

For example, to grant the owner all permissions, to grant group members read and update permissions, and to permit no access outside of the group, the permission mask should be defined as

```
(OPF_ALL | GPF_READ | GPF_WRITE | WPF_NONE)
```

If no permission mask is assigned to the file, then owner, group and world are granted all permissions. Permission masks, file ownership and group assignment may also be changed by using the **Security** function, or by the system administrator.

File Passwords

In addition to permission masks, files may be protected by **file passwords**. File passwords can be assigned at the time the file is created, by using the **Security** function, or by the system administrator. The file password is a 10 byte ASCIIZ string. A user cannot access the file, regardless of the file's permission mask, if the correct password is not supplied when the file is opened.

FairCom SQL Server

This section describes how to create a program that uses the FairCom SQL Server.

The FairCom SQL Server is a special variation of the FairCom Server. It provides advanced SQL access to any c-tree Plus database in addition to providing all of the functions found in FairCom Server.

13. SQL Introduction

The FairCom SQL Server is a special variation of the FairCom Server by FairCom corporation. It provides advanced SQL access to any c-tree Plus database. C language interface functions are provided to allow the software developer to access the database via **interactive** and **embedded** SQL calls.

This manual is not intended to be a tutorial on SQL or relational databases. We assume that the reader is familiar with at least the basic concepts. There are a number of reference books that can provide you with the knowledge you will need to be able to understand the FairCom SQL Server, including the book that accompanies this manual.

In addition, we assume that you are familiar with the basic concepts of the FairCom Server.

What is SQL?

SQL stands for Structured Query Language. Using SQL you are able to create and operate on **relational databases**, which are sets of related information stored in tables. A standard for SQL has been defined by ANSI (American National Standards Institute).

SQL allows flexible data insertion, deletion, updating, and selection without programming.

FairCom SQL Server

The FairCom SQL Server is a special version of the FairCom Server that supports all of the c-tree Plus features that have been discussed so far, and additionally supports the SQL features discussed in the next several chapters of this manual. Any program that works with the FairCom Server will work with the FairCom SQL Server **without any changes**. However, the reverse is not necessarily true. It is possible to write a program that works with the FairCom SQL Server that will not work with the FairCom Server. An error NSQL_ERR will be returned when you call one of the SQL functions when you are logged on to a non-SQL server.

Interactive vs. Embedded

There are two ways that you can make SQL calls - **interactive** and **embedded**..

Interactive

In an interactive mode you will be making inquiries to the FairCom SQL server in "real time". FairCom's **Natural Query** product can be used to enter SQL commands directly, with the results being shown immediately.

Embedded

When we speak of embedded SQL calls we are referring to entering SQL commands by function calls "embedded" or coded into a C language program. The functions for doing this are discussed in the **c-tree Plus Function Reference Guide**.

14. FairCom SQL Server Concepts

Tables and Views

$SColumns;SQL,ColumnsIn keeping with the ANSI SQL specification, files are referred to as *tables* and fields are referred to as *columns*. This syntax is intended to emphasize the fact that in some systems *tables* may not be physical files. Several *tables* may be in one file, or one *table* may be in several files. In the current implementations of the FairCom SQL Server, tables are either one data file and one index file, or several tables can be combined in one c-tree Plus **Superfile**.

The entries in a table are called *rows*. Each INSERT statement stores one or more new *rows* in the table.

A *view* is a *virtual table*. That is, it can be used as if it were a *table*, although in fact it is not. A *view* is used to:

- Restrict access to only some of the columns in a *table*;
- Restrict access to only some of the rows in a *table*;
- With the AUTHORIZATION mechanism, to restrict the ability to UPDATE to only some of the columns in a table;
- With the AUTHORIZATION mechanism, to restrict the ability to INSERT records;
- With the AUTHORIZATION mechanism, to restrict this ability to DELETE to only some of the rows in a table.

Note that a view may actually reference more than one base table. A view may also reference another view.

This mechanism, with the AUTHORIZATION mechanism described in the following section, allows flexible control of *tables* as well as who can do access them.

AUTHORIZATION Identifiers

Access to *tables* is controlled by an AUTHORIZATION identifier. By default, the AUTHORIZATION identifier is set to PUBLIC. The FairCom SQL Server maintains a "current" authorization identifier. To change the current identifier, there is a SET AUTHORIZATION statement. Certain statements also allow the AUTHORIZATION identifier to be specified.

An AUTHORIZATION identifier is associated with each *table* when the *table* is created. A user with this AUTHORIZATION identifier is the *table's* owner, and is allowed to perform all operations on that *table*. The GRANT statement is used

to allow other users to SELECT, INSERT, UPDATE, or DELETE rows from the *table*. With UPDATE, an AUTHORIZATION identifier can be granted update privileges for some (but not all) columns if necessary.

Privileges which are allowed with the GRANT statement can be disallowed using the REVOKE statement.

For many statements, the notation

```
authorization.table_name
```

can be used to completely specify a table. For example,

```
SELECT * FROM joe.s;
```

selects all records from table *s* created under authorization identifier *joe*. However, the current authorization identifier must have appropriate privelages for *joe.s* (in this case, SELECT privilege.

Transaction Processing

The FairCom SQL Server supports the ANSI SQL transaction methods as a part of the standard c-tree Plus transaction processing mechanism. These methods apply to INSERT, UPDATE, and DELETE.

There are two transaction commands: COMMIT and ROLLBACK. (The FairCom SQL Server also accepts the longer ANSI format: COMMIT WORK and ROLLBACK WORK.)

The concept of transaction processing involves the ability to cancel data modifications which are already complete. This is typically used to insure data integrity. For example, suppose there is a set of two tables representing invoices. One table is the invoice header, which might contain the customer number, date of invoice, number of items, etc., and which has one row for each invoice. The other table is the invoice items, which has one row for each item on the invoice. Thus there are many rows in the invoice item table for each row in the invoice header table. This is referred to as a "one to many" relation.

The invoice header and invoice item tables can get into an inconsistent situation. Suppose a set of invoice item rows are added into the invoice item table, and then the insertion into the invoice header table fails. There are then rows in the invoice item table for which no record exists in the invoice header table.

With transaction processing, the inserted rows of the invoice item table are not yet permanent. The rows can be eliminated using the ROLLBACK statement.

If all the related INSERTs are successfully completed, the data is made permanent using the COMMIT statement.

COMMIT means make INSERTs, UPDATEs, and DELETEs permanent. ROLLBACK means cancel all pending INSERTs, UPDATEs, and DELETEs. Unlike some SQL implementations, the FairCom SQL Server does not have an explicit statement to begin a transaction. Transaction processing is always in effect. The logical equivalent of a begin transaction statement can be achieved using a (possibly useless) ROLLBACK statement. A transaction begins with a COMMIT or ROLLBACK statement (or at the beginning of FairCom SQL Server processing), and ends with the next COMMIT or ROLLBACK statement.

The FairCom SQL Server transaction management system is designed to allow one FairCom SQL Server processor to manage asynchronous transactions from more than one user. The transaction file contains a user identifier which specifies the user who is associated with the data. When a COMMIT or ROLLBACK occurs, only the appropriate records are processed.

In the multi-user situation, FairCom SQL Server will only grant transaction access to a particular row to one user. If another user tries to UPDATE or DELETE the same row, FairCom SQL Server returns an error and does not complete the operation.

The FairCom SQL Server has a "serializability" flag which allows the user to enable or disable the serializability feature. Serializability is an ANSI feature which specifies certain behavior with regard to multi-user updates. The user should be careful not to enable the serializability feature for extended periods of time. In general, the serializability feature should not be used during interactive processing.

The user may wish to determine that a group of rows is available to him before he updates any of those rows. This can be accomplished by performing an UPDATE on each of those rows (the contents need not be changed). If any row is not available (because it is involved in another user's transaction), the user can release all the other rows using ROLLBACK. Otherwise, the user can UPDATE or DELETE the rows as desired, and then make the changes using COMMIT. When the serializability feature is enabled this is not necessary.

Error Codes

When an error occurs, the FairCom SQL Server sets values of three status variables. When using the interpreter, these values are displayed on the screen. When using the library, these values are available in global variables.

sql_err — **Error code**. Identifies the general error type. Values are negative except for 100, which means no records found. Values -3000 and below are errors which should not occur. Error codes -100 to -2999 are "normal" errors such as table not found, column name not recognized, etc.

err_pos — **Position code**. Identifies the specific location in code where Code error occurred. The position codes are unique.

uerr_cod

File code. This is the c-tree Plus error code returned by the Server. Note that this code may be non-zero even if *sql_err* is 0; this does not indicate an error. For example, *uerr_cod* will often contain a "key not found" or "end of file" status. If *sql_err* is 0, then ignore *uerr_cod*.

15. Using SQL with c-tree Plus™

The FairCom SQL Server provides the capability of SQL database access from within C programs. There are two methods for achieving this;

- **Embedded SQL**: The first method uses the standard ANSI SQL syntax for C programs and the c-tree Plus SQL preprocessor. The c-tree Plus SQL preprocessor replaces the ANSI SQL statements with C function calls. This is referred to as "embedded SQL" in the ANSI definition.
- **C Function Calls**: The programmer can also code the C function calls directly and avoid the use of the preprocessor. However, use of the standard calls and the C preprocessor provides a higher degree of portability to other SQL products.

The specification of the ANSI embedded SQL syntax is taken from American National Standards Institute publication ANSI X3.168-1989, titled Database Language - Embedded SQL.

Initializing c-tree Plus™

To be able to use the SQL function of the FairCom SQL Server you must first log on to the Server. The simplest way to do this is with the **InitCTree** function. After logging on you can do all of your file processing with SQL, or you can use both SQL and standard c-tree functions.

When your application is finished processing files, you must log off the Server. If you used **InitCTree** to log on, you must use **StopUser** to log off.

ANSI Embedded SQL C Program

A mixture of standard C statements and ANSI embedded SQL statements is known as an "embedded SQL C program". Generally you will place your embedded SQL statements in the same place in your program that a C statement would appear.

The format of the embedded SQL C program is;

```
< C statements >
Embedded SQL declaration section
< C statements >
Embedded SQL statements wherever appropriate
< C statements >
```

SQL Declaration section

A special section is defined for declaring variables. This section has the format;

```
EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL < host variable definition >;
    EXEC SQL < host variable definition >;
        . . . repeat as necessary
EXEC SQL END DECLARE SECTION;
```

This section declares variables that are visible to both the C code in the file and the embedded SQL statements in the file. The declaration section must be the first embedded SQL section in the file.

For example:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL char col1[17], col2[17];
EXEC SQL END DECLARE SECTION;
```

The variable type can also be a DEC or DATE_TIME structure. The DEC structure variable and the DATE_TIME structure variable are extensions to the ANSI specification, and allow data for DEC and DATE columns to be retrieved in its native format. Alternately, DEC data can be retrieved into float or double C variables, and DATE data can be retrieved into a C character array.

SQL Statements

An embedded SQL statement can appear in the same places that a C statement may appear. The SQL statement begins with the SQL prefix, which is:

```
EXEC SQL
```

and ends with the SQL terminator, with is either:

```
END-EXEC
```

or

```
;
```

An embedded SQL statement has the following format:

```
EXEC SQL
{
    < declare cursor > |
    < embedded exception declaration > |
    < SQL statement >
} ;
```

The *declare cursor* statement is a DECLARE CURSOR statement as defined in the **c-tree Plus Function Reference Guide.**

The *SQL statement* can be one of the standard SQL statements, described in the **c-tree Plus Function Reference Guide**, from the list below:

CLOSE	COMMIT	DELETE	FETCH
INSERT	OPEN	ROLLBACK	SELECT
UPDATE			

The *embedded exception declaration* specifies action to be taken if an SQL statement results in an error. The programmer can specify whether this applies to statements which result in error 100 (no records found). If more than one embedded exception declaration exists, then the most recently executed one is effective.

The form of the *embedded exception declaration* is:

```
WHENEVER < condition > < exception action >
```

condition can be SQLERROR, a test for any error, or NOT FOUND, a specific test for error 100 (no records found).

exception action can be CONTINUE, to allow the process to continue on, or GO TO *label*, where *label* is a regular C goto label.

For example;

```
EXEC SQL WHENEVER SQLERROR GOTO errorlabel;
```

Example Embedded SQL Program

The following example retrieves data from an SQL statement into local C character arrays. The two columns have SQL type CHARACTER length 15. The SQL table name is XYZ and authorization PUBLIC has SELECT privileges for the table.

```
#include "ctstdr.h"
#include "ctoptn.h"
#include "ctaerr.h"
#include "ctdecl.h"
#include "cterrc.h"
#include "ctsql.h"
```

```
main()
{
    if (InitCTree)
    {  printf("\nCannot initialize c-tree");
       exit(1);
    }

    EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL char col1[16], col2[16];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL DECLARE X CURSOR FOR SELECT * FROM XYZ;
    EXEC SQL OPEN X;
    EXEC SQL FETCH X INTO: col1,: col2;
    EXEC SQL CLOSE X;

    StopUser();
}
```

Additional Information

Some other details which are of interest are:

- The length of the C character array is one more than the size of the corresponding SQL column. That is, if the size (in the CREATE TABLE) statement is 10, the declaration in a C program is char[11].
- The SQL INTEGER type corresponds to the C long int type.
- The SQL SMALLINT type corresponds to the C short int type.
- The SQL REAL type corresponds to the C float type.
- The SQL DOUBLE PRECISION type corresponds to the C double type.
- The preprocessor creates an additional long int variable named SQLCODE. This variable contains the return code after the execution of each embedded SQL statement.
- The SQL DECIMAL type corresponds to the C structure type DEC defined in *ctsql.h*.
- The SQLDATE type corresponds to the C structure type DATE_TIME defined in *ctsql.h*.

C Function Call Interface

It is possible to write a program that does not require the C preprocessor. This section describes how to use SQL with C function calls. Note that the C preproces-

sor translates the embedded SQL C program file into a file containing only C function calls.

The three most important C functions are:

```
int sql1(char *sql_statement)
int sql2(char *sql_statement, int numvars, VARSPEC *vars)
int setvars(char *ptr, VARSPEC *vars, int type, int len, char *date_pattern)
```

- Function **sql1()** is used to invoke all SQL statements which do not return data.
- Function **sql2()** is used for SQL statements which return data, specifically FETCH and SELECT INTO.
- Function **setvars()** is used to initialize an array of data specifiers used by **sql2()** to determine the type and destination or returned data.

The following example retrieves data from an SQL statement into local C character arrays. The two columns have SQL type CHARACTER length 15. The SQL table name is XYZ and authorization PUBLIC has SELECT privileges for the table.

```
#include "ctstdr.h"
#include "ctoptn.h"
#include "ctaerr.h"
#include "ctdecl.h"
#include "cterrc.h"
#include "ctsql.h"
char col1[16];
char col2[16];
long SQLCODE;
VARSPEC vars[2];

main()
{
     if (InitCTree)
     {  printf("\nCannot initialize c-tree");
        exit(1);
     }

     SQLCODE =
        sql1("DECLARE X CURSOR FOR SELECT * FROM XYZ;");
     if (SQLCODE) /* do some error action */ ;
     SQLCODE = sql1("OPEN X:");
     if (SQLCODE) /* do some error action */ ;
     if (setvars(col1, &vars[0], CHARACTER,
         sizeof(col1), ((char*)0))
        /* do some error action */;
     if (setvars(col2, &vars[1], CHARACTER,
         sizeof(col2), ((char*)0))
```

```
        /* do some error action */;
    SQLCODE = sql2("FETCH X:", 2, vars);
    if (SQLCODE) /* do some error action */
    StopUser();
}
```

Additional **FETCH**es would retrieve data for additional records. The FairCom SQL Server provides alternate functions which fetch entire records rather than individual columns (see the r_fetch, r_insert, and r_update functions).

Output Modes

It is sometimes necessary to control the destination of FairCom SQL Server output data. Functions are provided for this purpose. FairCom SQL Server is in either "file output mode" or "buffer output mode". By default, the FairCom SQL Server is in file output mode and output is directed to standard output. Function **destbuf(char *address)** is used to place the FairCom SQL Server in buffer output mode. Function **destfil** returns the FairCom SQL Server to file output mode. Function **setdest(char *filename)** allows the destination file to be changed from standard output to any other file or device. After using **setdest**, if the mode is changed to buffer output mode, the destination file is not lost. Calling **destfil** will redirect output to that same file. The converse is not true. When **destfil** is called, the address of the buffer supplied in the preceding **destbuf** call is lost.

Appendices / Index

A. c-tree Plus™ Error Codes

There are two classes of errors: user errors and internal fatal errors. User errors arise because of application program errors (e.g., forgetting to open a file before using it), an unsuccessful operation (e.g., attempting to add a duplicate key value to an index not setup for duplicates), or the unavailability of system resources (e.g., no more disk space available). Internal fatal errors arise when c-tree data structures are clobbered (i.e., memory is overwritten), or when data files and/or index files are unexpectedly corrupted.

When a user error occurs, a global error code variable is set to an appropriate non-zero value usually referred to as the error code. For low level file functions, the error code variable is *uerr_cod*. For ISAM functions, the error code variable is *isam_err*. In addition, *isam_fil* is set to the file number of the data or index file involved.

A zero value for *uerr_cod* or *isam_err* is always associated with a successful file operation.

In addition to setting an error code variable, all of the ISAM functions and many of the low level file functions return the error code value themselves.

Please see Appendix B for the error codes for the SQL functions.

User Errors

Please note that a number of these "error" codes are not really errors, but rather indicators of an unsuccessful operation. For example, KDUP_ERR occurs if you attempt to add an existing entry to an index which does not support duplicate key values. This is not an error, but a situation to which your application program must be prepared to react.

Value	Symbolic Constant	Explanation
0	`NO_ERROR`	Successful operation.
2	`KDUP_ERR`	Key value already exists in index.
3	`KMAT_ERR`	Could not delete target key value since recbyt does not match associated data record position in index.
4	`KDEL_ERR`	Could not find target key value in index. No deletion performed. May indicate improper use of buffers during **ReWriteRecord** or **ReWriteVRecord**.
5	`KBLD_ERR`	Cannot call **DeleteKeyBlind** with an index which supports duplicate keys.

Value	Symbolic Constant	Explanation
6	BJMP_ERR	ctree() function jump table error.
10	SPAC_ERR	**InitCTree** parameters require too much space.
11	SPRM_ERR	Bad **InitCTree** parameter(s): either bufs < 3, idxs < 0, sect < 1, or dats < 0.
12	FNOP_ERR	Could not open file. Either file does not exist, filnam points to incorrect file name, or file is locked by another process.
13	FUNK_ERR	**OpenIFile** cannot determine type of file. Version 3.3 files must be rebuilt before using **OpenIFile**.
14	FCRP_ERR	File appears corrupt at open. This occurs if a file is updated; and the disk protocol is set at NOTFORCE; and **CloseIFile** is not executed. Rebuild file.
15	FCMP_ERR	Data file has been compacted (*ctcmpc*), but not cleared through CTRBLD. Rebuild data file (but do not force rebuild).
16	KCRAT_ERR	Could not create index file. Either no space is available on disk or filnam points to improper name.
17	DCRAT_ERR	Could not create data file. Either no space is available on disk or filnam points to improper name.
18	KOPN_ERR	Tried to create existing index file.
19	DOPN_ERR	Tried to create existing data file.
20	KMIN_ERR	Key length too large for node size. There must be room for at least 3 key values per node. The node size is given by sect * 128 where sect is 3rd **InitCTree** parameter.
21	DREC_ERR	Cannot create data file with record length smaller than 5.
22	FNUM_ERR	filno out of range: 0 < = filno < fils where fils is 2nd **InitCTree** parameter. This error may occur if c-tree has not been initialized (via **InitISAM**, **OpenISAM**, **CreateISAM** or **InitCTree**).
23	KMEM_ERR	Illegal index member number.
24	FCLS_ERR	Could not close file. Usually indicates that memory is clobbered.
25	KLNK_ERR	Bad link in deleted node list. Rebuild file.
26	FACS_ERR	File number (datno, keyno, or filno) is not in use.
27	LBOF_ERR	Data record position before 1st actual data record

Value	Symbolic Constant	Explanation
28	ZDRN_ERR	**AddKey** called with recbyt = 0.
29	ZREC_ERR	Data file routine called with recbyt = 0.
30	LEOF_ERR	recbyt exceeds logical end of file. If recbyt is correct, then data file header record may be incorrect. If so, rebuild data file.
31	DELFLG_ERR	Next record in delete chain of a fixed record length data file does not have 1st byte set to ff (hex). Data file header record may be corrupt; if so, rebuild data file. Or C255 constant is incorrect: see *ctcmpl.h*.
32	DDRN_ERR	Attempt to delete data record twice in a row.
33	DNUL_ERR	recptr is NULL in **ReadData**, **ReReadRecord**, **ReReadVRecord**, **WriteData** or **WriteVData**.
34	PRDS_ERR	Could not find correct predecessor node. Should not show up in a single user system. Indicates that an index insertion was interrupted before completion. Rebuild index.
35	SEEK_ERR	lseek failed in function ctio (*ctclib.c*). Possible causes are: out of disk space, corrupted record position in file, or corrupted file descriptor.
36	READ_ERR	read failed in function ctio (*ctclib.c*). Possible cause: corrupted data record position in file.
37	WRITE_ERR	write failed in function ctio. See 35 above.
38	VRTO_ERR	Could not convert a virtually opened file to an actually opened file. This might occur if your application uses up some file descriptors after a virtual file has been automatically closed. You can protect against this by lowering the MAXVFIL parameter in *ctoptn.h*.
39	FULL_ERR	The four byte data record position (or node position) address space has been exhausted.
40	KSIZ_ERR	sect * 128 (where sect is 3rd **InitCTree** parameter) was larger when index was created. Buffers are too small for nodes.
41	UDLK_ERR	Could not unlock data record. If dummy lock file is in use, be sure it has a file mode of 3.
42	DLOK_ERR	Could not obtain data record lock. In a DOS 3.1 system, be sure that the network is up

Value	Symbolic Constant	Explanation
43	FVER_ERR	Current configuration parameters are inconsistent with the configuration parameters at the time of file creation.
44	OSRL_ERR	Data file serial number overflow
45	KLEN_ERR	Index key length exceeds MAXLEN parameter. Change MAXLEN in *ctoptn.h* and recompile c-tree.
46	FUSE_ERR	File number is already in use
47	FINT_ERR	c-tree has not been initialized
48	FMOD_ERR	A function has been called for the wrong type of file: e.g., a variable record length function is called for a fixed record length data file.
49	FSAV_ERR	Could not write file directory updates to disk during file extension.
50	LNOD_ERR	Could not lock index file node.
51	UNOD_ERR	Could not unlock index file node. If a dummy lock file is in use, be sure it has a file mode of 3.
52	KTYP_ERR	Variable length and/or floating point keys disabled in *ctoptn.h*.
53	FTYP_ERR	Variable length records disabled in *ctoptn.h*
54	REDF_ERR	Attempt to write a read only file
55	DLTF_ERR	Could not delete file. Has another process gained control of the file?
56	DLTP_ERR	filno must reference a file opened with an EXCLUSIVE file mode.
57	DADV_ERR	Attempt to write or delete a data record without the proper lock, or in conflict with another lock.
58	KLOD_ERR	**LoadKey** called with incorrect key number. You cannot continue.
59	KLOR_ERR	**LoadKey** called with key out of order. You may skip this key & continue.
60	KFRC_ERR	60
61	CTNL_ERR	NULL fcb detected during I/O.
62	LERR_ERR	File must be opened exclusively.
63	RSER_ERR	start file / lof file serial number error.

Value	Symbolic Constant	Explanation
64	RLEN_ERR	file name too long in log file.
65	RMEM_ERR	not enough memory during recovery.
66	RCHK_ERR	log file entry failed to find checkpoint.
67	RENF_ERR	could not rename file.
68	LALC_ERR	could not allocate block for control list.
69	BNOD_ERR	node does not belong to index.
70	TEXS_ERR	transaction already pending.
71	TNON_ERR	no active transaction.
72	TSHD_ERR	no space for shadow buffer.
73	TLOG_ERR	LOGFIL encountered during shadow only.
74	TRAC_ERR	recovery: two active tran for user.
75	TROW_ERR	recovery: bad tran owner.
76	TBAD_ERR	recovery: bad tran type.
77	TRNM_ERR	recovery: file name to long.
78	TABN_ERR	transaction abandoned: too many log extents.
79	FLOG_ERR	could not log file opn/cre/cls/del.
80	BKEY_ERR	NULL target or bad keyno.
81	ATRN_ERR	transaction allocation error.
82	UALC_ERR	user allocation error.
83	IALC_ERR	isam allocation error.
84	MUSR_ERR	maximum users exceeded.
85	WCNF_ERR	wrtnod conflict with transaction.
86	DEAD_ERR	dead lock detected.
87	QIET_ERR	system not quiet: files in use.
88	LMEM_ERR	linked list memory allocation error.
89	TMEM_ERR	memory allocation during tran processing.
90	NQUE_ERR	could not create queue.
91	QWRT_ERR	queue write error.
92	QMRT_ERR	queue memory error during write.

Value	Symbolic Constant	Explanation
93	QRED_ERR	queue read error.
94	PNDG_ERR	pending error: cannot save or commit tran.
95	STSK_ERR	could not start task
96	LOPN_ERR	start-file/log open error.
97	SUSR_ERR	bad user handle.
98	BTMD_ERR	bad transaction mode.
99	TTYP_ERR	transaction type / mode conflict.
100	ICUR_ERR	No current ISAM record for data file isam_fil.
101	INOT_ERR	Could not satisfy ISAM search request for index isam_fil.
102	INOD_ERR	ISAM parameter file does not exist. Be sure that filnam parameter points to correct name. isam_fil value is undefined.
103	IGIN_ERR	Could not read ISAM parameter file Initialization record. isam_fil is undefined. Be sure that parameter file is not empty, and that the correct short integer input conversion character has been specified in *ctoptn.h*.
104	IFIL_ERR	Too many data files and indices in ISAM parameter file. Change MAXFIL in *ctoptn.h*. isam_fil is undefined.
105	IUND_ERR	Could not undo a rejected ISAM update. Data file must be rebuilt. (During the rebuild, look for records with rejected duplicate key values.)
106	IDRI_ERR	Could not read ISAM parameter file Data File Description record for data file number isam_fil. Be sure that parameter file is consistent with RTREE setting in *ctoptn.h*.
107	IDRK_ERR	Too many indices for data file number isam_fil in ISAM parameter file. Change MAX_DAT_KEY in *ctoptn.h*.
108	IMKY_ERR	keyno for index file member out of sequence. keyno must equal host index file keyno plus member number.
109	IKRS_ERR	Too many key segments defined in ISAM parameter file for index number isam_fil. Change MAX_KEY_SEG in *ctoptn.h*. Be sure that parameter file is consistent with RTREE setting in *ctoptn.h*.

Value	Symbolic Constant	Explanation
110	ISRC_ERR	Could not read ISAM parameter file Key Segment Description record for index number isam_fil.
111	IKRI_ERR	Could not read ISAM parameter file Index File Description record. isam_fil indicates the relative index number for an unspecified data file.
112	IPND_ERR	**LockISAM(ENABLE)** found pending locks. To help enforce two-phase lock protocol
113	INOL_ERR	No space left in c-tree's internal lock table. Change MAX_LOCKS in *ctoptn.h*.
114	IRED_ERR	1st byte of fixed length data record found by ISAM routine equals delete flag.
115	ISLN_ERR	Sum of key segment lengths does not match key length for index number isam_fil.
116	IMOD_ERR	Bad mode parameter.
117	IMRI_ERR	Could not read ISAM parameter file Index Member record.
118	SKEY_ERR	**NextInSet** or **PreviousInSet** called with a keyno which does not match keyno in last call to **FirstInSet** or **LastInSet**.
119	SKTY_ERR	**FirstInSet** called for numeric key type.
120	RRLN_ERR	Not enough dynamic memory for record buffer in *ctrbld.d*.
121	RSPC_ERR	Not enough space for rebuild sort buffer area. Change SORT_SPACE in *ctrbld.c*.
122	RMOD_ERR	Attempt to change between fixed and variable length records during rebuild. Once a data file is created, its record length characteristic cannot be changed.
123	RVHD_ERR	A variable length data record is not preceded by a valid record mark. The file is apparently corrupted.
124	INIX_ERR	Number of indices in index file does not match IFIL structure in call to **OpenIFile**.
125	IINT_ERR	c-tree is already initialized via a previous call to **InitISAM**, **OpenISAM**, **CreateISAM** or **InitCTree**.
126	ABDR_ERR	Bad directory path get
127	ARQS_ERR	Could not send request
128	ARSP_ERR	Could not receive answer

Value	Symbolic Constant	Explanation
129	NINT_ERR	c-tree not initialized.
130	AFNM_ERR	NULL file name pointer in **OpenIFile**.
131	AFLN_ERR	File name length exceeds message size
132	ASPC_ERR	No room for application message buffer
133	ASKY_ERR	Could not identify server
134	ASID_ERR	Could not get servers message id
135	AAID_ERR	Could not allocate application id
136	AMST_ERR	Could not get application message status
137	AMQZ_ERR	Could not set application message size
138	AMRD_ERR	Could not get rid of application message
139	ABNM_ERR	Badly formed file name
140	VMAX_ERR	Variable record length exceeds 65,529 bytes.
141	AMSG_ERR	Required message size exceeds maximum
142	SMXL_ERR	Application MAXLEN > server's MAXLEN *ctoptn.h*.
143	SHND_ERR	Communications handler not installed
144	QMEM_ERR	Application could not id output queue
145	ALOG_ERR	No message space. Was login ok?
146	VDLK_ERR	Could not update available space information in variable record length data file.
147	VDLFLG_ERR	Record pointed to by available space information is not marked deleted in variable record length data file.
148	VLEN_ERR	Attempt to write a variable length record into a file position which is too small for the record.
149	VRLN_ERR	Variable record length passed to **AddVRecord** is less than minimum record length established at file creation.
150	SHUT_ERR	Server is shutting down
151	STRN_ERR	Could not shut down; transactions pending
152	LEXT_ERR	Could not extend logfile.
153	VBSZ_ERR	Buffer in call to **ReReadVRecord** is too small for the variable length record.

Value	Symbolic Constant	Explanation
154	VRCL_ERR	Attempt to read (**ReReadVRecord**) a zero length record from a variable record length data file.
155	SYST_ERR	Native system failure.
158	VFLG_ERR	Attempt to read (**ReReadVRecord**) a record not marked active.
159	VPNT_ERR	Zero data record position in variable record length function.
160	ITIM_ERR	Multi-user interference: key value changed between index search and subsequent data record read.
161	SINA_ERR	User appears inactive
162	SGON_ERR	Server has gone away.
163	SFRE_ERR	No more room in server lock table
164	SFIL_ERR	File number out of range
165	SNFB_ERR	No c-tree file control block
166	SNMC_ERR	No more c-tree file control blocks in server
167	SRQS_ERR	Could not read request
168	SRSP_ERR	Could not send answer
169	TCRE_ERR	Create file already opened (in recovery).
170	SFUN_ERR	Bad function number at server
171	SMSG_ERR	Application message size exceeds server size
172	SSPC_ERR	Could not allocate server message buffer
173	SSKY_ERR	Could not identify server
174	SSID_ERR	Could not get server message id
175	SAMS_ERR	Server could not allocate user message area
176	SMST_ERR	Could not get server message status
177	SMQZ_ERR	Could not set message server message size
178	SINM_ERR	Unexpected file # assigned to [si] in rcv.
179	SOUT_ERR	Server is at full user capacity
180	IKRU_ERR	Could not read r-tree symbolic index name.
181	IKMU_ERR	Could not get dynamic memory for r-tree symbolic index name.

Value	Symbolic Constant	Explanation
182	IKSR_ERR	Cannot accommodate temporary r-tree files. Increase MAXFIL.
183	IDRU_ERR	Could not read r-tree data field symbolic names.
184	ISDP_ERR	Multiple set buffer space already allocated.
185	ISAL_ERR	Not enough dynamic memory for multiple sets.
186	ISNM_ERR	Set number out of range in **ChangeSet**.
187	IRBF_ERR	Null buffer pointer in r-tree.
188	ITBF_ERR	Null target buffer pointer in r-tree.
189	IJSK_ERR	JOINS_TO skip condition in r-tree.
190	IJER_ERR	JOINS_TO error condition in r-tree.
191	IJNL_ERR	JOINS_TO null fill condition in r-tree.
192	IDSK_ERR	IS_DETAIL_FOR skip condition in r-tree.
193	IDER_ERR	IS_DETAIL_FOR error condition in r-tree.
194	IDNL_ERR	IS_DETAIL_FOR null fill condition in r-tree.
195	IDMU_ERR	Could not get dynamic memory for r-tree data field symbolic names.
196	ITML_ERR	Exceeded RETRY_LIMIT on error 160 in r-tree.
197	IMEM_ERR	Could not get memory for IFIL block.
198	BIFL_ERR	Improper IFIL block.
199	NSCH_ERR	key segment refers to schema but no schema is defined.
400	RCRE_ERR	resource already enabled
401	RNON_ERR	resources not enabled
402	RXCL_ERR	file must be exclusive to enable res
403	RZRO_ERR	empty resource id
404	RBUF_ERR	output buffer to small
405	RDUP_ERR	resource id already added
406	RCSE_ERR	bad resource search mode
407	RRED_ERR	attempt to get non-resource info
408	RNOT_ERR	resource not found
410	USTP_ERR	user not active

Value	Symbolic Constant	Explanation
411	BSUP_ERR	not a superfile
412	SOPN_ERR	attempt to create open superfile member
413	SDIR_ERR	superfile host not opened
414	SNST_ERR	cannot nest superfiles
415	SADD_ERR	illegal **AddKey** to superfile
416	SDEL_ERR	illegal **DeleteKeyBlind** to superfile
420	TPND_ERR	key update with pending transaction
421	BTFL_ERR	filter not supported yet. You probably have an invalid value in the *mode* parameter of your **DoBatch** call.
422	BTFN_ERR	function not supported. You probably have an invalid value in the *mode* parameter of your **DoBatch** call.
423	BTIC_ERR	incomplete batch. You specified BAT_COMPLETE in a **DoBatch** call
424	BTAD_ERR	add list error. This is an internal processing problem. You may have a memory overwrite of some sort.
425	BTIP_ERR	batch already in progress. You made a call to **DoBatch** to open a new batch before you have completed the prior one. Make a call with a *mode* of BAT_CAN first.
426	BTNO_ERR	no batch active. You are calling **DoBatch** with a mode of BAT_CAN or BAT_NXT
427	BTST_ERR	status info already returned. You are making two consecutive **DoBatch** calls looking just for status information
428	BTMT_ERR	no more info
429	BTBZ_ERR	*bufsiz* is too small for a single record. You need a larger buffer for this file.
430	BTRQ_ERR	*request* is a NULL pointer.
432	FLEN_ERR	fixed length string requires len in DODA.
433	SSCH_ERR	segment definition inconsistent with schema.
434	DLEN_ERR	very long def block not supported.
435	FMEM_ERR	file def memory error.
436	DNUM_ERR	bad def number.
437	DADR_ERR	defptr NULL during GETDEFBLK.

Value	Symbolic Constant	Explanation
438	DZRO_ERR	Resource not founnd.
440	DDDM_ERR	dynamic dump already in progress
441	DMEM_ERR	no memory for dynamic dump file buffer
442	DAVL_ERR	one or more files not available for dump
443	DSIZ_ERR	file length discrepancy
444	DCRE_ERR	could not create file during dump rcv
445	SDAT_ERR	not enough data to assemble key.
446	BMOD_ERR	invalid key mode.
447	BOWN_ERR	you must be the owner of this file.
448	DEFP_ERR	File definition permission denied.
450	LUID_ERR	invalid user id when logging on server
451	LPWD_ERR	invalid password when logging on server
452	LSRV_ERR	server could not process user/acct info
453	NSRV_ERR	invalid server name
455	SGRP_ERR	user does not belong to group
456	SACS_ERR	group access denied
457	SPWD_ERR	file password invalid
458	SWRT_ERR	write permission not granted
459	SDLT_ERR	delete permission denied
460	SRES_ERR	resource not enabled
461	SPER_ERR	bad permission flag
500	SQLINIT_ERR	
501	SQLCONNECT_ERR	
502	SQL_REQUEST_ERROR	
503	SQL_INVALID_CONTINUE	
504	NSQL_ERR	Not using an SQL Server. You are making an SQL call but you are not using the FairCom SQL Server.

Internal Fatal Errors

The following error codes should not occur during proper operation of the c-tree Plus routines. Their main function is to help find programming bugs, especially if you modify the c-tree Plus code. Our experience shows that almost all occurrences of an internal error are caused by memory overwrites from the application code.

Error Number	Explanation
200	Neither LOW_HIGH nor HIGH_LOW defined in *ctoptn.h*.
201	Both LOW_HIGH and HIGH_LOW defined in *ctoptn.h*.
202	Neither NOTFORCE, FPUTONLY nor FPUTFGET defined in *ctoptn.h*.
203	More than one of NOTFORCE, FPUTONLY and FPUTFGET were defined in *ctoptn.h*.
206	Update flag inconsistency between index file header and buffer status information
207	Corrupt node found in nodser function
208	Updated (but not written) node found in FPUTFGET disk I/O mode.
209	nodpnt routine called for a duplicate leaf node
210	Undefined key type found in compar
211	Negative number of key values indicated in node
212	No leaf node found during **FirstKey** operation
213	No leaf node found during **LastKey** operation
214	Corrupt tree found in fndkey
215	No leaf node found in fndkey
216	Attempt to save node with negative number of key values
217	Attempt to transfer key values between buffers for two different index files
218	Corrupt tree found in **AddKey**
219	No leaf node found in **AddKey**
220	Corrupt tree found in **DeleteKey**

Error Number	Explanation
221	No leaf node found in **DeleteKey**.
222	Undefined file access flag found: filacs member of file control structure must be set to 'y' (active), 'v' (active, but temporarily closed), or 'n' (not active)
225	VARLDATA not defined in *ctoptn.h*. Select VARLDATA option and recompile *ctaddk.c*.
226	Undefined key segment translation mode
227	While extending the size of a fixed record length data file, delete flags could not be written into new (but unused) records at the end of the file.
228	Expected deleted record mark not found while collapsing consecutive deleted areas
230	B-Tree node has conflicting member number value during write operation
231	B-tree node has conflicting member number value during read operation
232	Attempt to expand non-existent compressed key value
233	Illegal value for key compression bytes
235	Illegal comparison value during node insert
236	Illegal key value shift detected with compressed keys
237	Attempt to get an index node at byte offset zero
238	Illegal key value shift detected with compressed keys
239	variable key expansion request with NO_VARLK
240	exceeded maximum b-tree levels. Increase MAXLEV or increase number of node sectors
968	multiple unlock request error
969	combined operation error
970	error establishing block: existing block
974	unblocking unknown block type
975	error establishing unknown block type
976	error unblocking file extension operation
977	error unblocking write operation
978	error unblocking read operation

Error Number	Explanation
979	error checking for blocking condition
980	server lock list error
981	server lock list error
982	server lock list error
983	server lock list error
984	could not add lock to server lock list
992	chnacs in **CreateIndexFile** at server
993	illegal chnacs in **CreateDataFile** at server
994	illegal chnacs in **OpenFile** at server for index members
996	illegal chnacs in **OpenFile** at server
997	getctf function cannot find available FCB
998	getctm function cannot find a set of FCB's to link

B. SQL Error Codes

Two digit errors are syntax errors. Three and four digit errors are run time errors.

Standard Errors

Code	Description
0	no error
1	syntax error. The first word of the statement was not recognized.
2	Select list error. The select list is the section of a **SELECT** statement between **SELECT** and **FROM**. For example, this error is caused by **SELECT SNO, FROM S;** because the comma requires another field before **FROM**.
3	**FROM** keyword not found. After **SELECT** and the select list, the next token was not **FROM**. For example, **SELECT SNO S;**.
4	A left parenthesis (did not follow **INSERT INTO** able_name.
5	Tokens following **WHERE** keyword do not form a valid search condition. For example, **SELECT SNO FROM S WHERE XYZ XYZ**.
6	In a **GROUP BY** clause, the word **BY** did not follow the word **GROUP**.
7	A **GROUP BY** column name is invalid. For example, **SELECT * FROM S GROUP BY 123**.
8	In an **ORDER BY** clause, the word **BY** did not follow the word **ORDER**.
9	In a **CREATE SCHEMA** statement, the word **SCHEMA** did not follow the word **CREATE**.
10	In a **CREATE SCHEMA** statement, the word **AUTHORIZATION** did not follow **CREATE SCHEMA**.
11	Invalid authorization identifier in a **CREATE SCHEMA** statement.
12	The first word of a schema element (table or view) description was not **CREATE** or the termination symbol ;.
13	**INSERT** was not followed by **INTO**.
14	Invalid table name. For example, **INSERT INTO 123**.
15	Missing right parenthesis after **INSERT INTO** *table_name* (*column_name* [,*column_name*]. *For example,* **INSERT INTO S (SNO VALUES ('S1').**

Code	Description
16	**VALUES** missing in **INSERT INTO** statement. For example, **INSERT INTO S (SNO) ('S1').**
17	(missing after **VALUES** in **INSERT INTO** statement. For example, **INSERT INTO S (SNO) VALUES 'S1.**
18	) missing after **INSERT INTO** value list. For example, **INSERT INTO S (SNO) VALUES ('S1';.**
19	Bad **INSERT INTO** value. For example, **INSERT INTO S (SNO) VALUES (APPLES).**
20	Invalid cursor name. For example, **DECLARE 123 CURSOR FOR ...**
21	**DELETE** not followed by **FROM.**
22	Invalid deletion table name. For example, **DELETE FROM 123 ...**
23	Deletion table name not followed by **WHERE.** For example, **DELETE FROM S SNO = 'S1';.**
24	**CURRENT** not followed by **OF.**
25	**SET** missing in **UPDATE** statement. For example, **UPDATE S SNO = 'S1'...** instead of **UPDATE S SET SNO = 'S1'...**
26	Invalid **UPDATE** column name. For example, **UPDATE 123...**
27	= missing from **UPDATE** set clause. For example, **UPDATE S SET SNO 'S1'.**
28	= not followed by valid expression or **NULL.**
100	No records found. In interpreter, displays the message **No matching records.**
101	In an **INSERT INTO** statement, the value corresponding to a CHARACTER field is not a character string.
102	In an **INSERT INTO** statement, a character string value is associated with a column whose type is not CHARACTER.
103	In an **INSERT INTO** statement, a value is real, but the corresponding field type is DECIMAL, INTEGER, or SMALLINT.
104	In an **INSERT INTO** statement, a character string is associated with a FLOAT or DOUBLE PRECISION field.
105	In an **INSERT INTO** statement, a value is not provided for a field with the **NOT NULL** attribute.
106	In an **INSERT INTO** statement, the value specified for a UNIQUE field is a duplicate of a value already in the database.

Code	Description
107	In an **INSERT INTO** statement, the value specified for a UNIQUE field is a duplicate of a value already in the database (but not yet COMMITTed).
108	In an **INSERT INTO** using a **SELECT** statement, the effective authorization identifier does not have select privilege for one or more files in the **SELECT** statement.
109	In an **INSERT INTO** using a **SELECT** statement, the number of fields to be inserted does not match the number of fields in the **SELECT** statement.
110	In an **INSERT INTO** using a **SELECT** statement, data from a numeric column cannot be loaded into a character column.
111	In an **INSERT INTO** using a **SELECT** statement, data from a character column cannot be loaded into a numeric column.
112	In an **INSERT INTO** using a **SELECT** statement, data from a column of type FLOAT or DOUBLE cannot be loaded into a column of type SMALLINT, INTEGER, or NUMERIC.
113	In an **INSERT INTO** using a **SELECT** statement, no records were found to insert.
200	In a **CREATE TABLE** statement, a field name in the unique list is not recognized.
201	In a **CREATE TABLE** statement, a data file with the same name already exists.
202	In a **CREATE TABLE** statement, an index file with the same name already exists.
203	Dictionary write error during **CREATE TABLE**. This is not caused by a duplicate table name (see error 204), but by some other condition. One possible cause is a full disk.
204	A **CREATE TABLE** statement specified a table name which is already in the file dictionary. Note that no data or index file exists, because errors 201 and 202 did not occur. A likely cause of error 204 is copying the dictionary files from one directory to another, but neglecting to copy all the data (or index) files.
205	During **CREATE TABLE**, an error was detected when writing to the file dictionary.
206	Dictionary write error during **CREATE VIEW**. This is not caused by a duplicate table name (see error 207), but by some other condition. The most likely cause is lack of space on the disk.

Code	Description
207	A **CREATE VIEW** statement specified a table name which is already in the file dictionary.
300	A table name was not recognized. The name may be misspelled, or may be unknown to the system.
301	No records were found in the field dictionary for a table.
302	An error occurred reading the field dictionary.
303	An expected key was not found in the key dictionary.
304	An error occurred reading the key dictionary.
400	A field name is not recognized. The field name is displayed, and is not recognized for any files mentioned in the statement.
401	A NULL field was used in a query.
500	A character string was used in a numeric expression in a select list.
600	A character string was used in a numeric expression in a **WHERE** clause.
601	A field of type CHARACTER was used in a numeric expression in a **WHERE** clause.
700	**FETCH** requested for cursor whose name has not been declared.
701	**FETCH** requested for closed cursor.
702	**FETCH** requested for cursor positioned past last row.
703	A **FETCH** statement discovered that there are no more rows available.
800	**OPEN** requested for cursor which has not been declared.
801	**OPEN** requested for cursor which is already open.
802	**CLOSE** requested for cursor which has not been declared.
803	**CLOSE** requested for cursor which is not open.
804	Positioned **DELETE** requested for cursor which has not been declared.
805	Cursor referenced in positioned **DELETE** refers to more than one file.
806	Positioned **DELETE** requested for cursor which is not open.
807	Positioned **DELETE** requested for cursor which is at beginning or end of file.
808	Table name in positioned **DELETE** is different from table name in associated **DECLARE CURSOR**.

Code	Description
809	A **DECLARE ? CURSOR** statement requested one more cursor than the maximum allowable number. Note that this refers to distinct cursor names. The same cursor name may be assigned to different cursors at different times during program execution, thus providing a way of working around the limit which causes this error.
900	Sub-query of =,,, =,, or did not find any values. This type of sub-query must find exactly one value.
901	Sub-query of =, , =, , or found more than one value. This type of sub-query must find exactly one value.
1000	CHARACTER field or constant used in numeric expression.
1001	CHARACTER field or constant used in numeric expression.
1002	CHARACTER field or constant used in numeric expression.
1003	CHARACTER field or constant used in numeric expression.
1004	CHARACTER field or constant used in numeric expression.
1005	CHARACTER field or constant used in numeric expression.
1006	CHARACTER field or constant used in numeric expression
1100	Positioned **UPDATE** requested for cursor which has not been declared.
1101	Positioned **UPDATE** requested for cursor which refers to more than one file.
1102	Positioned **UPDATE** requested for cursor which is not open.
1103	Positioned **UPDATE** requested for cursor which is at beginning or end of file.
1104	**UPDATE** statement file name does not match **DECLARE CURSOR** file name.
1105	Character string assigned to numeric field in **UPDATE** statement.
1106	Numeric value assigned to CHARACTER field in **UPDATE** statement.
1107	Fractional (real) value assigned to integer field in **UPDATE** statement.
1108	Real value assigned to CHARACTER, SMALLINT, INTEGER, or DECIMAL field in **UPDATE** statement.
1109	**UPDATE** statement changes value which produces conflict with UNIQUE constraint on table.
1110	**UPDATE** statement changes value (not yet COMMITted) which produces UNIQUE conflict.
1200	In a **GRANT** statement, too many user names were specified.

Code	Description
1201	In a **GRANT** statement, the table name is not known to the system.
1202	In a **GRANT** statement, the user does not have the required privileges.
1210	For a **SELECT** statement, the user does not have the required select privileges.
1211	For an **INSERT** statement, the user does not have the required insert privilege.
1212	For a **DELETE** statement using a cursor, the user does not have the required delete privilege.
1213	For a searched **DELETE** statement, the user does not have the required delete privilege.
1214	For an **UPDATE** statement using a cursor, the user does not have the required update privilege.
1215	For an **UPDATE** statement using a cursor, the user does not have permission to update the requested columns.
1216	For a searched **UPDATE** statement, the user does not have the required update privilege.
1217	For a searched **UPDATE** statement, the user does not have permission to update the requested columns.
1300	An **INSERT** statement names a read-only table.
1301	A **DELETE** statement using a cursor names a read-only table.
1302	A searched **DELETE** statement names a read-only table.
1303	An **UPDATE** statement using a cursor names a read-only table.
1304	A searched **UPDATE** statement names a read-only table.
1400	The row associated with an **UPDATE** statement is involved in another transaction.
1401	The row associated with a **DELETE** statement is involved in another transaction.
1402	Internal error during deletion. Should not occur in normal operation. A pending transaction record could not be deleted (DELETE following INSERT without intervening COMMIT).

Internal Errors

Errors with numbers above 3000 indicate that an internal error occurred or a capacity constraint was exceeded. The position code identifies a unique location in the code for each error.

3000	Error writing to index file. File system error code identifies the particular error.
3001	Too many files. The maximum number of files which may be in use is set by compile-time constant **CTFILS**.
3002	Error opening file. File system error code identifies particular error.
3003	Bad field type. Internal error which should not occur.
3004	Error creating index file. File system error code identifies the particular error.
3005	Predicate stack overflow. Can be caused by complex mathematical expressions. Stack size can be increased by increasing compile time constant **MAXPRS**.
3006	Parse tree node limit exceeded. Can be caused by complex mathematical expressions. Size of parse tree can be increased by increasing compile time constant **MAXPXS**.
3007	Stack underflow. Position code can be used to identify stack. Internal error which should not occur.
3008	Operation code error. Position code can be used to identify place in source code where error occurred. Internal error which should not occur.
3009	Error closing file. File system error code identifies the particular error.
3010	Too many links. The maximum number of links can be increased by increasing compile time constant **MAXLNK**.
3011	Bad type in PPNODE or PNODE table in ptbl1.c-ptbl6.c. Internal error which should not occur.
3012	Too many entries in value table. The maximum number of value table entries may be increased by increasing compile time constant **MAXVAL**.
3013	Too many sub-queries. The maximum number of sub_queries may be increased by increasing compile time constant **MAXSET**.
3014	Cursor status error. Internal error which should not occur.
3015	Bad intermediate code value type. Internal error which should not occur.

3016	Function class error. Internal error which should not occur.
3017	Stack position error. Internal error which should not occur.
3018	Boolean operation on character data.
3019	Memory allocation error. A memory allocation request failed. The position code indicates where the error occurred.
3020	Too many set files. The maximum number of set files may be increased by increasing compile time constant **MAXODS**.
3021	Too many constants. The maximum number of constants may be increased by increasing compile time constant **MAXTBL**.
3022	Too many identifiers. The maximum number of identifiers may be increased by increasing compile time constant **MAXIDS**.
3023	Tried to use non-character constant as escape character. Internal error which should not occur.
3024	Transaction command code error. Internal error which should not occur.
3025	Data/index sync error. A record could not be read from the data file at the position specified by the index file. Internal error which should not occur.
3026	Error allocating new record for data file. Unless the disk is full, this error should never occur.
3027	Error adding key to index file.
3028	Error writing to data file.
3029	Variable length flag error. The first byte of a variable length record is not an appropriate mark.
3030	Too many fields. Controlled by compile time constant **MAXFTB**.
3031	Too many intermediate code instructions. Controlled by compile time constant **MAXICD**.
3032	View dictionary sync error. Information expected in the view dictionaries was not found. May indicate damaged view dictionary files.
3033	A key known to be in an index file was not located.
3034	A deletion failed for a key known to be in an index file.
3035	A data record could not be freed for reuse.
3036	Error adding record for union. Internal error which should not occur.
3037	Commit deletion error.
3038	Commit read error.

3039	Commit insertion error.
3040	Key length exceeds data record length.
3041	Bad predicate type.
3042	Data file creation error.
3043	During a FETCH or SELECT INTO operation using embedded library support, **outbuf** was NULL. This is probably caused by failure to set **outbuf** using the **destbuf** function.
3044	In a call to **cql2**, a required pointer was NULL. Usually caused by incorrectly setting pointer within structure **VARSPEC**.
3045	Error decoding FETCH value in **cql2**.
3046	A type encountered by **cql2** is unrecognized.
3047	FETCH output data did not begin with BEG_MARK.
3048	Data expected in dictionaries not found.
3049	Data expected in dictionaries not found.
3050	Scanner error.
3051	Too many string constants. To increase the maximum number of string constants, increase the value of compile time constant **MAXTBL**.
3052	Error occurred while inserting data into temporary file. Most likely cause is a full disk.
3053	Transaction file open error.
3054	Transaction master key to binary conversion error. Internal error which should never occur.
3055	Record deletion error.
3056	Character function with numeric or date fields.
3057	Link resolution error.
3058	Input buffer length exceeded.
3059	FETCHP requested for C-tree variable length file.
3060	Could not open input file.
3061	FETCHP requested for cursor with HAVING. This is not supported.
3062	Report/Screen Format File error.
3063	Report format editing error.
3064	DATE/TIME decoding error.
3065	Type mismatch in DATE/TIME function.

3066	Too many header entries in report. This can occur because of multi-line headings.
3067	Attempted to drop a table which has active view associated with it. You must drop all the views before dropping the table.
3068	Dictionary error during drop.
3069	DROP TABLE requested for view.
3070	Internal error. Note position code and file system error code. Call for help.
3071	Record addition error.
3072	Column name not recognized in ALTER TABLE.
3073	Too many databases.
3074	Database name not recognized
3075	SELECT INTO used with interpreter. This is not supported.
3076	For a view created WITH CHECK OPTION, an INSERT or UPDATE does not satisfy the view condition.

C. Function Name Cross Reference

Each c-tree function (with the exception of the SQL functions) has two names that can be used, the **full name** and the **abbreviated name**. The full names are used throughout this manual, and are fairly descriptive of the action that the function is to perform. The abbreviated name is shorter and more cryptic. You will see it if you examine the c-tree source code. Prior versions of the c-tree File Handler used the abbreviated names. If you have a program that worked with older versions of c-tree you will not have to change the function calls over to the full names.

Full Names

This table lists the functions alphabetically by the Full Name.

Full Name	Abbreviated Name
Abort	TRANABT
AddKey	ADDKEY
AddRecord	ADDREC
AddResource	ADDRES
AddVRecord	ADDVREC
AllocateSet	ALCSET
Begin	TRANBEG
ChangeSet	CHGSET
ClearTranError	TRANCLR
CloseFile	CLSFIL
CloseIFile	CLIFIL
CloseISAM	CLISAM
CloseRFile	CLRFIL
Commit	TRANEND
CreateDataFile	CREDAT
CreateDataFileXtd	CREDATX
CreateIFile	CREIFIL
CreateIFileXtd	CREIFILX

Full Name	Abbreviated Name
CreateIndexFile	CREIDX
CreateIndexFileXtd	CREIDXX
CreateIndexMember	CREMEM
CreateISAM	CREISAM
CreateISAMXtd	CREISAMX
CurrentFileOffset	GETCURP
CurrentISAMKey	GETCURK
DeleteFile	DELFIL
DeleteKey	DELCHK
DeleteKeyBlind	DELBLD
DeleteRecord	DELREC
DeleteResource	DELRES
DeleteVRecord	DELVREC
DoBatch	BATSET
EnableResource	ENARES
EstimateKeySpan	ESTKEY
FirstInSet	FRSSET
FirstKey	FRSKEY
FirstRecord	FRSREC
FreeSet	FRESET
GetAlternateSequence	GETALTSEQ
GetDODA	GETDODA
GetFileInfo	GETFIL
GetGTEKey	GTEKEY
GetGTKey	GTKEY
GetGTERecord	GTEREC
GetIFile	GETIFIL
GetKey	EQLKEY
GetLTEKey	LTEKEY

Full Name	Abbreviated Name
GetLTERecord	LTEREC
GetLTKey	LTKEY
GetRecord	EQLREC
GetResource	GETRES
GetSerialNbr	SERIALNUM
GetSymbolicNames	GETNAM
GetTempFileName	TMPNAME
InitCTree	INTREE
InitCTreeXtd	INTREEX
InitISAM	INTISAM
InitISAMXtd	INTISAMX
KeyAtPercentile	FRCKEY
LastInSet	LSTSET
LastKey	LSTKEY
LastRecord	LSTREC
LoadKey	LOADKEY
LockData	LOKREC
LockISAM	LKISAM
NewData	NEWREC
NewVData	NEWVREC
NextInSet	NXTSET
NextKey	NXTKEY
NextRecord	NXTREC
NbrOfKeyEntries	IDXENT
NbrOfRecords	DATENT
OpenFile	OPNFIL
OpenFileXtd	OPNFILX
OpenIFile	OPNIFIL
OpenIFileXtd	OPNIFILX

Full Name	Abbreviated Name
OpenISAM	OPNISAM
OpenISAMXtd	OPNISAMX
OpenFileWithResource	OPNRFIL
OpenFileWithResourceXtd	OPNRFILX
PositionSet	MIDSET
PreviousInSet	PRVSET
PreviousKey	PRVKEY
PreviousRecord	PRVREC
PutDODA	PUTDODA
ReadData	REDREC
ReadVData	RDVREC
RebuildIFile	RBLIFIL
RebuildIFileXtd	RBLIFILX
ReleaseData	RETREC
ReleaseVData	RETVREC
ReReadRecord	RRDREC
ReReadVRecord	REDVREC
ResetRecord	UPDCURI
RestoreSavePoint	TRANRST
ReWriteRecord	RWTREC
ReWriteVRecord	RWTVREC
Security	SECURITY
SetAlternateSequence	SETALTSEQ
SetRecord	SETCURI
SetSavePoint	TRANSAV
SetVariableBytes	SETVARBYTS
StopUser	STPUSR
TransformKey	TFRMKEY
UpdateFileMode	PUTFIL

Full Name	Abbreviated Name
UpdateResource	UPDRES
VDataLength	GTVLEN
VRecordLength	GETVLEN
WriteData	WRTREC
WriteVData	WRTVREC

Abbreviated Names

This table lists the functions alphabetically by the Abbreviated Name.

Abbreviated Name	Full Name
ADDKEY	AddKey
ADDREC	AddRecord
ADDRES	AddResource
ADDVREC	AddVRecord
ALCSET	AllocateSet
BATSET	DoBatch
CHGSET	ChangeSet
CLIFIL	CloseIFile
CLISAM	CloseISAM
CLRFIL	CloseRFile
CLSFIL	CloseFile
CREDAT	CreateDataFile
CREDATX	CreateDataFileXtd
CREIDX	CreateIndexFile
CREIDXX	CreateIndexFileXtd
CREIFIL	CreateIFile
CREIFILX	CreateIFileXtd
CREISAM	CreateISAM
CREISAMX	CreateISAMXtd
CREMEM	CreateIndexMember

Abbreviated Name	Full Name
DATENT	NbrOfRecords
DELBLD	DeleteKeyBlind
DELCHK	DeleteKey
DELFIL	DeleteFile
DELREC	DeleteRecord
DELRES	DeleteResourcex
DELVREC	DeleteVRecord
ENARES	EnableResource
EQLKEY	GetKey
EQLREC	GetRecord
ESTKEY	EstimateKeySpan
FRCKEY	KeyAtPercentile
FRESET	FreeSet
FRSKEY	FirstKey
FRSREC	FirstRecord
FRSSET	FirstInSet
GETALTSEQ	GetAlternateSequence
GETCURK	CurrentISAMKey
GETCURP	CurrentFileOffset
GETDODA	GetDODA
GETFIL	GetFileInfo
GETIFIL	GetIFile
GETNAM	GetSymbolicNames
GETRES	GetResource
GETVLEN	VRecordLength
GTEKEY	GetGTEKey
GTEREC	GetGTERecord
GTKEY	GetGTKey
GTVLEN	VDataLength

Abbreviated Name	Full Name
IDXENT	NbrOfKeyEntries
INTISAM	InitISAM
INTISAMX	InitISAMXtd
INTREE	InitCTree
INTREEX	InitCTreeXtd
LKISAM	LockISAM
LOADKEY	LoadKey
LOKREC	LockData
LSTKEY	LastKey
LSTREC	LastRecord
LSTSET	LastInSet
LTEKEY	GetLTEKey
LTEREC	GetLTERecord
LTKEY	GetLTKey
MIDSET	PositionSet
NEWREC	NewData
NEWVREC	NewVData
NXTSET	NextInSet
NXTKEY	NextKey
NXTREC	NextRecord
OPNIFILX	OpenIFileXtd
OPNISAM	OpenISAM
OPNISAMX	OpenISAMXtd
OPNFIL	OpenFile
OPNFILX	OpenFileXtd
OPNRFIL	OpenFileWithResource
OPNRFILX	OpenFileWithResourceXtd
PRVKEY	PreviousKey
PRVREC	PreviousRecord

Abbreviated Name	Full Name
PRVSET	PreviousInSet
PUTDODA	PutDoda
PUTFIL	UpdateFileMode
RBLIFILX	RebuildIFileXtd
RDVREC	ReadVData
REDREC	ReadData
REDVREC	ReReadVRecord
RETREC	ReleaseData
RETVREC	ReleaseVData
RRDREC	ReReadRecord
RWTREC	ReWriteRecord
RWTVREC	ReWriteVRecord
SECURITY	Security
SERIALNUM	GetSerialNbr
SETALTSEQ	SetAlternateSequence
SETCURI	SetRecord
STPUSR	StopUser
SETVARBYTS	SetVariableBytes
TFRMKEY	TransformKey
TMPNAME	GetTempFileName
TRANABT	Abort
TRANBEG	Begin
TRANCLR	ClearTranError
TRANEND	Commit
TRANRST	RestoreSavePoint
TRANSAV	SetSavePoint
UPDCURI	ResetRecord
UPDRES	UpdateResource
WRTREC	WriteData

Abbreviated Name	Full Name
WRTVREC	WriteVData

D. Record Schemas

The c-tree Plus file handler is a record oriented file manager. However, to support intelligent key value operations, and to support higher level systems applications such as report generation, user interfaces and SQL, c-tree provides support for individual field definitions.

The definition of each field comprising a record can be stored in the data file itself if the data file has not disabled RESOURCE support. The d-tree component of the FairCom ToolBox and the FairCom SQL Server will automatically store the record schema in the data file, or you can do it yourself by calling the function **PutDODA**. The advantage of storing the schema is that applications can use this information without any outside information. Further, c-tree Plus can use field numbers to specify key segments if the schema is stored in the file, enabling new and more flexible key segment definitions.

This appendix discusses the structure of the Record Schema Resource, how it can be used to define Key Segments, and how you can create this resource via low level c-tree function calls.

Record Schema

The Record Schema consists of a **schema map header**, followed by a number of **schema field blocks**, which are in turn followed by a number of **schema name details**. The structure of this information is provided here for your information. You will not have to create this information directly, as it is prepared automatically by either d-tree, the FairCom SQL Server, or via a call to the **PutDODA** function that is described later.

Schema Map

A schema map is stored in the form of a map header which is defined by the *ConvMap* typedef shown below;

```
typedef struct convmap {
    UTEXT   flavor;     /* 1-68000  2-8086  3-pdp */
    UTEXT   align;      /* 1-byte   2-word  4-double word */
    UTEXT   flddelm;    /* field delimiter byte */
    UTEXT   padding;    /* field padding byte */
    VRLEN   maplen;     /* total length of map including header */
    VRLEN   nbrflds;    /* number of fields in map */
    VRLEN   nbrblks;    /* number of field blocks (which may be
                        less than nbrflds due to repeat counts */
    } ConvMap;
```

The map header is followed by a specified (*nbrblks*) number of field blocks defined by the *ConvBlk* typedef shown below;

```
typedef struct convblk {
    UCOUNT  len;          /* field length / or maximum length */
    UTEXT   kind;         /* c-tree Plus field type (see later) */
    UTEXT   repcnt;       /* repeat count */
    } ConvBlk;
```

A *repcnt* zero means no repeat fields; a repeat count of 1 means one more field with the same definition, etc.

Schema Name Details

Each field can be assigned a symbolic name. The names (which are not counted in the *maplen* parameter) associated with each field are stored consecutively in the form

1 byte length count (which includes length count and terminator)
n byte name
1 byte null terminator

Therefore, the symbolic name "FIRST" will be stored as

7FIRST0

If no name is associated with a field, then a single byte of value 1 is stored. For example, if the first and third fields are named "FIRST" and "THIRD", respectively, and the second field has no name, the names will be stored as

7FIRST017THIRD0

There is a single name for each field block. If there is a non-zero repeat count, only one name is stored. There is not a name stored for each repeated field.

Record Structure

While c-tree Plus can support virtually any organization of data fields, there are some guidelines which promote maximum support by c-tree Plus and FairCom's other products. We strongly recommend following these guidelines unless there are compelling reasons not to do so.

The field types above can be grouped into fixed and varying length fields. For this grouping, the fixed refers to whether or not the space occupied by the field is fixed, not whether the contents are fixed.

When defining your record structure, the varying length fields will either have a known maximum length or be unlimited (within the bounds of the length count bytes if any). For example, you may have a varying length description field which is null terminated and which is guaranteed not to be longer than 1024 bytes. This is a varying length field with a known maximum. Conversely, you may have a 2 byte length count field which is varying in length, but there is no specified maximum length. This is a varying length field without a known maximum. Therefore, each field can be classified into one of three groups:

- Fixed Length
- Varying Length Max
- Varying Length No Max

We recommend that fields be placed in the record in the order shown above.

Field Types

The *kind* member of the *ConvBlk* structure refers to a **c-tree Plus field type**. c-tree Plus recognizes the following field types:

- signed and unsigned char's
- signed and unsigned 2 and 4 byte integers
- single, double and extended precision floats
- scaled and unscaled BCD numbers
- arbitrary fixed length data
- fixed and variable length strings delimited by a specified field delimiter
- fixed and variable length strings whose actual contents are specified by 1, 2 or 4 byte length counts which occur at the beginning of the field.

The field delimiter defaults to a null byte. However, the **SetVariableBytes** function can be used to change the delimiter to a value other than the default.

Note that the 1, 2 and 4 byte length counts include the length count bytes themselves. Therefore, the smallest legitimate value for a 1 byte length count is 1.

A fixed length field with varying length contents means that the field occupies a fixed number of bytes in the record structure, but that a field delimiter or a length count specifies the actual amount of data in the field.

The constants used to represent these field types (found in *ctport.h*) are:

CT_CHAR	signed char
CT_CHARU	unsigned char
CT_INT2	signed two byte integer
CT_INT2U	unsigned two byte integer
CT_INT4	signed four byte integer
CT_INT4U	unsigned four byte integer
CT_MONEY	signed four byte integer interpreted as number of pennies
CT_DATE	unsigned four byte integer interpreted as date
CT_TIME	unsigned four byte integer interpreted as time
CT_SFLOAT	four byte floating point
CT_DFLOAT	eight byte floating point
CT_SQLBCD	ten byte scaled BCD (includes exponent)
CT_EFLOAT	greater than eight byte floating point (not supported as a key segment)
CT_BCD	unscaled BCD
CT_BCDMONEY	unscaled BCD interpreted as number of pennies
CT_ARRAY	arbitrary fixed length data
CT_FSTRING	fixed length field delimited data
CT_FPSTRING	fixed length data with 1 byte length count
CT_F2STRING	fixed length data with 2 byte length count
CT_F4STRING	fixed length data with 4 byte length count
CT_STRING	varying length field delimited data
CT_PSTRING	varying length data with 1 byte length count
CT_2STRING	varying length data with 2 byte length count
CT_4STRING	varying length data with 4 byte length count

Create Record Schema with PutDODA

The definition of each of the fields comprising a record can be stored in the data file itself if the data file has not disabled RESOURCE support. The advantage of storing the schema is that applications can use this information without any outside information. Further, c-tree Plus can use field numbers to specify key segments if

the schema is stored in the file, enabling new and more flexible key segment definitions.

An application can store the schema in a file directly by use of the **PutDODA** function. A **DODA** is a *data object definition array*. Each element of the array is comprised of a structure of type **DATOBJ** which is typedef'ed in *ctport.h*. Only three of the first four fields of the DATOBJ are required for c-tree Plus (The r-tree Report Generator and the d-tree Development System require the first four fields).

DATOBJ is defined as follows:

```
typedef struct {
    TEXT   *fsymb;  /* ptr to optional symbol name     */
    TEXT   *fadr;   /* adr of field in record buffer   */
    UCOUNT  ftype;  /* type indicator                  */
    UCOUNT  flen;   /* field length                    */
        ...
    } DATOBJ;
```

fsymb points to an optional symbolic name for the field (*fsymb* should be NULL if no name is desired). *fadr* is not used by c-tree Plus (its value is ignored). *ftype* is one of the field types specified in the table on page D-4. *flen* is set to the field's length for fixed length fields, or the known maximum for varying length fields with a known maximum length, or zero for varying length fields without known maximum. If the field type has an intrinsic length (which is true for types CT_CHAR through CT_SQLBCD), then a zero length will be automatically replaced by the intrinsic length.

Given a data record with structure;

```
struct {
    COUNT   delflg;          /* delete flag         */
    TEXT    zipcode[10];     /* zipcode             */
    LONG    ssn;             /* social security #   */
    TEXT    name[50];        /* name                */
} DATA_FORMAT;
```

The corresponding DODA would be defined as

```
DATOBJ doda[] = {
   {"DeleteFlag",NULL,CT_INT2U},
   {"ZipCode",NULL,CT_FSTRING,10},
   {"SocialSecurity",NULL,CT_INT4U},
   {"Name",NULL,CT_STRING,50}
   };
```

Note that *zipcode* is considered to take up a fixed space in the record (CT_FSTRING), while *name* takes up a variable amount of space (CT_STRING), up to a maximum of 50 bytes.

See the function reference for **PutDODA** for details concerning its use.

For the r-tree Report Generator, the second field of the DATOBJ structure must be filled-in, and the DODA is terminated by an entry of the form;

```
{NULL,NULL,0,0,-1}.
```

A DODA in this form can also be used by **PUTDODA**, but the terminating entry shown above is not counted as one of the fields (which is the last argument to **PutDODA**).

Key Segments

As described in the section beginning on page 5-14, ISAM level applications define key values by specifying where to find, and how to treat each data field (or part of a data field) comprising the key. Each part of a key is called a key segment.

Once you have defined a Record Schema for a file you can use it to easily define key segments.

When key segment mode SCHSEG (12) or any of its varients (USCHSEG, VSCHSEG, or UVSCHSEG) are used, the interpretation of the data is based on the underlying type found in the schema. Based on the above DODA example, a triple of the form

```
segment position = 2
segment length   = 4
segment mode     = 12 (SCHSEG)
```

implies use the third field, social security number (the first field is schema field zero). Since it is an unsigned integer, it will be treated as such by the index routines.

The use of VSCHSEG instead of SCHSEG is rather subtle. If a fixed length field, such as the *zipcode* field in the above DODA example, does have a field delimiter or a length count, then specifying VSCHSEG indicates that if the actual data is shorter than the segment length, pad the contents out to the full segment length. The default padding is an ASCII space character, but the **SetVariableBytes** function can be used to change the padding byte. If SCHSEG is used instead, then delimiters or length counts will not be used to determine how to pad the key segment. Whatever is in the fixed length field will be used; even if it causes "garbage" to be used beyond the actual contents. Of course, if a fixed length field is carefully prefilled or padded, then the use of SCHSEG will not cause any problem.

The segment mode value can be modified to permit the use of alternative collating sequences or to collate the key segment in descending order. Note that these modifications apply to the segment, not the entire key. "OR"ing DSCSEG into the segment mode (or adding 16 to the segment mode) indicates descending order.

"OR"ing ALTSEG into the segment mode (or adding 32 to the segment mode) indicates the use of an alternative collating sequence. Both of these modifications can be used at the same time.

Alternative collating sequences are assigned to an index file with the **SetAlternate-Sequence** function.

E. Common Entry Point Functions

c-tree Plus offers two calling sequences for each function: the traditional c-tree call by function name in which each operation is called by a separate function, and the common entry point in which all functions can be accessed through the **ctree()** function. For example, the **AddKey** function can be called in either of these two ways:

```
AddKey(keyno,target,recbyt,typadd);
```

or

```
ctree(FN_ADDKEY,keyno,target,&recbyt,NULL,NULL,typadd);
```

Either approach will produce the same results. Some developers may feel that the common entry point approach better suits their approach to programming.

If you are using the FairCom Server you will notice that the common entry point approach is what the client side of c-tree Plus uses to send the function calls to the server. The traditional functions are just remapped to common entry point functions. If you use the common entry point function in your application, without any traditional functions, your program will not have to link in the remapping code. Therefore, if program memory is limited, you may want to use the common entry point approach.

If you are producing a single user program using the FairCom Server linked directly to your application you may find that this memory saving is critical.

Extended Feature Parameter Blocks

Virtually all of the **extended features** (such as file security) are passed to c-tree as part of a new set of parameter blocks defined in *ctparm.h*. Any application which uses the single entry point **ctree()** function must include *ctparm.h*. This include also contains the function numbers (FN_ADDKEY, FN_ADDREC, ...) used in the **ctree()** function.

The definitions of the parameter blocks are shown blow. They are referenced as parameters in the function definitions listed at the end of this appendix.

Please note the following values assigned to the various character array sizes for the parameter block definitions:

Symbolic Constant	Value	Use (including null terminator)
EXZ	8	Incremental IFIL data and index file name extensions
IDZ	16	User and Group ID's
PWZ	10	User and file passwords
DSZ	64	Symbolic Field and Index Names
FNZ	255	File and Server Names

Logon Block

```
typedef struct logblk {
    COUNT    files;     /* number of files & indices */
    COUNT    userprof;  /* user profile mask */
    COUNT    ibuffers;  /* # index buffers */
    COUNT    dbuffers;  /* # of data buffers */
    COUNT    nsectors;  /* # of sectors per buffer */
    COUNT    reservd1;  /* reserved */
    VRLEN    reservd2;  /* reserved */
    TEXT     userid[IDZ];   /* user id */
    TEXT     userword[PWZ]; /* user password */
    TEXT     servname[FNZ]; /* optional server name */
    } LOGBLK;
```

The *ibuffers*, *dbuffers* and *nsectors* members are ignored when connecting to a FairCom Server. The *userid* and *userword* members are ignored in non-server operation. The *userprof* member controls whether or not to perform automatic **TransformKey**'s on targets sent to ISAM level search routines. Set *userprof* to USERPRF_NTKEY if you wish to disable the automatic **TransformKey**. The *servname* member supports multiple server environments.

ISAM Block

```
typedef struct isamblk {
    LOGBLK   isamlog;   /* login block */
    LONG     permmask;  /* CreateISAM permission mask */
    TEXT     groupid[IDZ];  /* CreateISAM Group id */
    TEXT     fileword[PWZ]; /* file password */
    TEXT     parmname[FNZ]; /* parameter file name */
    } ISAMBLK;
```

The *isamlog* member is simply a logon block interpreted as above. The optional *permmask* and *groupid* members are only used by **CreateISAM** to set the files' permission masks and passwords. The *fileword* member is used by **CreateISAM** to

set the password and by **OpenISAM** to access the file. The *parmname* member holds the parameter file name.

IFIL Block

```
typedef struct ifilblk {
    LONG      permmask;  /* CreateIFile permission mask */
    TEXT      groupid[IDZ];  /* CreateIFile Group id */
    TEXT      fileword[PWZ]; /* file password */
    TEXT      dataextn[EXZ]; /* data extension */
    TEXT      indxextn[EXZ]; /* indx extension */
    IFIL     *ptr_ifil;  /* IFIL pointer */
    } IFILBLK;
```

The *permmask* and *groupid* members hold the optional security information assigned at file creation. *fileword* holds the optional password assigned at creation or the password passed at a subsequent file open. The *dataextn* and *indxextn* members hold optional file name extensions for the data and index files, respectively. If a file name extension member begins with a null byte, then the default extension is used, ".dat" for data files and ".idx" for index files. To signify that no extension should be added, pass the member containing only blanks (and a null terminator). (Both file name extension members cannot be blank since the data file and index must have unique names.) The *ptr_ifil* member points to a traditional IFIL structure.

Create File Block

```
typedef struct creblk {
    UCOUNT   length;     /* record / key length */
    UCOUNT   extdsize;   /* file extent size */
    COUNT    filemode;   /* file mode */
    COUNT    keytyp;     /* index key type */
    COUNT    keydup;     /* index duplicate flag */
    COUNT    member;     /* index member # */
    LONG     permmask;   /* permission mask */
    TEXT     groupid[IDZ];  /* group id */
    TEXT     fileword[PWZ]; /* file password */
    TEXT     filename[FNZ]; /* file name */
    } CREBLK;
```

The *length* member specifies the record length for data files or the key length for index files. The *extdsize* member contains the file size extension parameter, and the *filemode* member contains the file mode to use at creation. The *keytyp*, *keydup* and *member* members of the parameter block are ignored for data files and represent the key type, key duplicate and index member number, respectively, for index files. The *permmask*, *groupid* and *fileword* members hold the optional file security information. Finally, *filename* contains the file name to be used at create.

Open File Block

```
typedef struct openblk {
    COUNT    filemode;  /* file mode */
    TEXT     fileword[PWZ]; /* file password */
    TEXT     filename[FNZ]; /* file name */
    } OPENBLK;
```

The *filemode* member holds the file mode to be used at open time. The *fileword* member contains the optional password, and the *filename* member contains the file name to be used at open.

Key Estimate Block

```
typedef struct estmblk {
    TEXT    key1[MAXLEN];    /* 1st key */
    TEXT    key2[MAXLEN];    /* 2nd key */
    } ESTMBLK;
```

The Key Estimate Block is used with the ESTKEY function which estimates the number of key values ocurring between *key1* and *key2*.

Function Calls

The single entry point calling sequence is based on the following function prototype:

```
    COUNT ctree(COUNT fn,COUNT filno,VOID *ptr1,LONG
*plong,
     VOID *ptr2,VRLEN *psize,COUNT mode);
```

where *fn* is the function number, *filno* is usually a file number, *ptr1* and *ptr2* point to various types of data or parameter blocks, *plong* points to a long integer, *psize* points to a long integer used to convey or return the size of an object, and *mode* is a short integer is for various specifications.

Because of the manner in which C functions pass parameters, it is not necessary to supply all seven parameters all of the time. Generally, you need only supply enough parameters to satisfy the particular function. BUT, you cannot skip parameters.

Note that c-tree always returns an error code. If the traditional function does not return an error code, then a return value is passed back as one of the parameters to c-tree. For example, **GetKey** returns a long value corresponding to the record postion which contains the target key value. In the traditional c-tree approach, we call

```
recpos = GetKey(keyno,target);
```

With the ctree() form, we call

```
errcod = ctree(FN_EQLKEY,keyno,target,&recpos);
```

recpos will be passed back as an output parameter of **ctree()**. In the following listing, *retval* is used to denote the parameter in which the return value of the function is returned. (c-tree Plus functions which return an error code do not use a *retval* parameter.)

Each of the c-tree Plus functions are listed below with their traditional call by name form and the ctree(FN_...) form. Note that the **extended functions** (those that have a suffix of **Xtd**) use the same single entry point call as the equivalent standard function. Also - the function numbers are in the form of the **short function name**, as listed in Appendix C.

```
COUNT    Abort()
    ctree(FN_TRANABT)

COUNT    AddKey(keyno,target,recbyt,typadd)
    ctree(FN_ADDKEY,keyno,target,&recbyt,NULL,NULL,typadd)

COUNT    AddRecord(datno,recptr)
    ctree(FN_ADDREC,datno,recptr)

COUNT    AddResource(datno,resptr,varlen)
    ctree(FN_ADDRES,datno,resptr,NULL,NULL,&varlen)

COUNT    AddVRecord(datno,recptr,varlen)
    ctree(FN_ADDVREC,datno,recptr,NULL,NULL,&varlen)

COUNT    AllocateSet(numset)
    ctree(FN_ALCSET,numset)

LONG     Begin(mode)
    ctree(FN_TRANBEG,mode,NULL,&retval)

COUNT    ChangeSet(setnum)
    ctree(FN_CHGSET,setnum)

COUNT    ClearTranError()
    ctree(FN_TRANCLR)

COUNT    CloseFile(filno,filmod)
    ctree(FN_CLSFIL,filno)

COUNT    CloseIFile(ifilptr)
    ctree(FN_CLIFIL,0,&ifilblk)
    /*
```

```
        ** ifilblk is defined above under the discussion of the
        ** IFIL Block found in CTPARM.H.
        */

COUNT   CloseISAM()
    ctree(FN_CLISAM)

COUNT   CloseRFile(datno)
    ctree(FN_CLRFIL,datno)

COUNT   Commit(mode)
    ctree(FN_TRANEND,mode)

COUNT   CreateDataFile(datno,filnam,datlen,xtdsiz,filmod)
        also-
COUNT   CreateDataFileXtd(datno,filnam,datlen,xtdsiz,filmod)
    ctree(FN_CREDAT,datno,&createblk)
    /*
    ** createblk is defined above under the discussion of the
    ** Create Block found in CTPARM.H.
    */

COUNT   CreateIFile(ifilptr)
        also-
COUNT   CreateIFileXtd(ifilptr)
    ctree(FN_CREIFIL,0,&ifilblk)
    /*
    ** ifilblk is defined above under the discussion of the
    ** IFIL Block found in CTPARM.H.
    */

COUNT   CreateIndexFile(keyno,filnam,keylen,keytyp,keydup,nomemb,
                        xtdsiz,filmod)
        also-
COUNT   CreateIndexFileXtd(keyno,filnam,keylen,keytyp,keydup,
          nomemb,xtdsiz,filmod)
    ctree(FN_CREIDX,keyno,&createblk)
    /*
    ** isamblk is defined above under the discussion of the
    ** ISAM block found in CTPARM.H.
    */

COUNT   CreateIndexMember(keyno,keylen,keytyp,keydup,membno)
    ctree(FN_CREMEM,keyno,&createblk)
    /*
    ** isamblk is defined above under the discussion of the
    ** ISAM block found in CTPARM.H.
    */
```

```
COUNT    CreateISAM(filnam)
         -also
COUNT    CreateISAMXtd(filnam)
     ctree(FN_CREISAM,0,&isamblk)
     /*
     ** isamblk is defined above under the discussion of the
     ** ISAM block found in CTPARM.H.
     */

TEXT *   CurrentISAMKey(keyno,idxval,plen)
     ctree(FN_GETCURK,keyno,idxval,NULL,plen)

LONG     CurrentFileOffset(datno)
     ctree(FN_GETCURP,datno,NULL,&retval)

COUNT    DELFIL(filno)
     ctree(FN_DELFIL,filno)

COUNT    DeleteKey(keyno,target,recbyt)
     ctree(FN_DELCHK,keyno,target,&recbyt)

LONG     DeleteKeyBlind(keyno,target)
     ctree(FN_DELBLD,keyno,target,&retval)

COUNT    DeleteRecord(datno)
     ctree(FN_DELREC,datno)

COUNT    DeleteResource(datno,resptr)
     ctree(FN_DELRES,datno,resptr)

COUNT    DeleteVRecord(datno)
     ctree(FN_DELVREC,datno)

COUNT    DoBatch(filno,request,bufptr,bufsiz,mode)
     ctree(FN_BATSET,filno,request,NULL,bufptr,&bufsiz,mode)

COUNT    EnableResources(datno)
     ctree(FN_ENARES,datno)

LONG     EstimateKeySpan(keyno,idxval1,idxval2)
     ctree(FN_ESTKEY,keyno,&estm,&retval)

COUNT    FirstInSet(keyno,target,recptr,siglen)
     ctree(FN_FRSSET,keyno,target,NULL,recptr,NULL,siglen)

LONG     FirstKey(keyno,idxval)
     ctree(FN_FRSKEY,keyno,idxval,&retval)

COUNT    FirstRecord(filno,recptr)
```

```
        ctree(FN_FRSREC,filno,recptr)

    COUNT    FreeSet()
        ctree(FN_FRESET)

    COUNT    GetAlternateSequence(keyno,altseq)
        ctree(FN_GETALTSEQ,keyno,altseq)

    VRLEN    GetDODA(datno,bufsiz,bufptr,mode)
        ctree(FN_GETDODA,datno,bufptr,&bufsiz,NULL,&retval,mode)

    LONG     GetFileInfo(filno,mode)
        ctree(FN_GETFIL,filno,NULL,&retval,NULL,NULL,mode)

    LONG     GetGTEKey(keyno,target,idxval)
        ctree(FN_GTEKEY,keyno,target,&retval,idxval)

    COUNT    GetGTERecord(keyno,target,recptr)
        ctree(FN_GTEREC,keyno,target,NULL,recptr)

    LONG     GetGTKey(keyno,target,idxval)
        ctree(FN_GTKEY,keyno,target,&retval,idxval)

    VRLEN    GetIFile(datno,bufsiz,bufptr)
        ctree(FN_GETIFIL,datno,bufptr,&bufsiz,NULL,&retval)

    LONG     GetKey(keyno,target)
        ctree(FN_EQLKEY,keyno,target,&retval)

    LONG     GetLTEKey(keyno,target,idxval)
        ctree(FN_LTEKEY,keyno,target,&retval,idxval)

    COUNT    GetLTERecord(keyno,target,recptr)
        ctree(FN_LTEREC,keyno,target,NULL,recptr)

    LONG     GetLTKey(keyno,target,idxval)
        ctree(FN_LTKEY,keyno,target,&retval,idxval)

    COUNT    GetRecord(keyno,target,recptr)
        ctree(FN_EQLREC,keyno,target,NULL,recptr)

    LONG     GetResource(datno,resptr,bufptr,bufsiz,resmode)
        ctree(FN_GETRES,datno,resptr,&retval,bufptr,&bufsiz,resmode)

    LONG     GetSerialNbr(datno)
        ctree(FN_SERIALNUM,datno,NULL,&retval)

    COUNT    GetSymbolicNames(filno,nambuf,buflen,mode)
        ctree(FN_GETNAM,filno,nambuf,NULL,NULL,&buflen,mode)
```

```
COUNT    GetTempFileName(bufptr,bufsiz)
    ctree(FN_TMPNAME,0,bufptr,NULL,NULL,&bufsiz)

COUNT    InitISAM(bufs,fils,sect)
         Also-
COUNT    InitISAMXtd(bufs,fils,sect)
    ctree(FN_INTISAM,0,&logonblk)
    /*
    ** logonblk is defined above under the discussion of the
    ** Logon Block found in CTPARM.H.
    */

COUNT    InitCTree(bufs,fils,sect)
         Also-
COUNT    InitCTreeXtd(bufs,fils,sect)
    ctree(FN_INTREE,0,&logonblk)
    /*
    ** logonblk is defined above under the discussion of the
    ** Logon Block found in CTPARM.H.
    */

LONG     KeyAtPercentile(keyno,idxval,fractal)
    ctree(FN_FRCKEY,keyno,idxval,&retval,NULL,NULL,fractal)

COUNT    LastInSet(keyno,target,recptr,siglen)
    ctree(FN_LSTSET,keyno,target,NULL,recptr,NULL,siglen)

LONG     LastKey(keyno,idxval)
    ctree(FN_LSTKEY,keyno,idxval,&retval)

COUNT    LastRecord(filno,recptr)
    ctree(FN_LSTREC,filno,recptr)

COUNT    LoadKey(keyno,target,recbyt,typadd)
    ctree(FN_LOADKEY,keyno,target,&recbyt,NULL,NULL,typadd)

COUNT    LockISAM(lokmod)
    ctree(FN_LKISAM,lokmod)

COUNT    LockRecord(datno,lokmod,recbyt)
    ctree(FN_LOKREC,datno,NULL,&recbyt,NULL,NULL,lokmod)

LONG     NbrOfKeyEntries(keyno)
    ctree(FN_IDXENT,keyno,NULL,&retval)

LONG     NbrOfRecords(datno)
    ctree(FN_DATENT,datno,NULL,&retval)
```

```
LONG     NewData(datno)
    ctree(FN_NEWREC,datno,NULL,&retval)

LONG     NewVData(datno,varlen)
    ctree(FN_NEWVREC,datno,NULL,&retval,NULL,&varlen)

COUNT    NextInSet(keyno,recptr)
    ctree(FN_NXTSET,keyno,recptr)

LONG     NextKey(keyno,idxval)
    ctree(FN_NXTKEY,keyno,idxval,&retval)

COUNT    NextRecord(filno,recptr)
    ctree(FN_NXTREC,filno,recptr)

COUNT    OpenFile(filno,filnam,filmod)
         Also-
COUNT    OpenFileXtd(filno,filnam,filmod)
    ctree(FN_OPNFIL,filno,&openblk)
    /*
    ** openblk is defined above under the discussion of the
    ** Open Block found in CTPARM.H.
    */

COUNT    OpenFileWithResource(filno,filnam,filmod)
         Also-
COUNT    OpenFileWithResourceXtd(filno,filnam,filmod)
    ctree(FN_OPNRFIL,filno,&openblk,&retval)
    /*
    ** openblk is defined above under the discussion of the
    ** Open Block found in CTPARM.H.
    **
    ** retval is a long integer in which the file number
    ** is returned
    */

COUNT    OpenIFile(ifilptr)
         Also- COUNT OpenIFileXtd(ifilptr)
    ctree(FN_OPNIFIL,0,&ifilblk)
    /*
    ** ifilblk is defined above under the discussion of the
    ** IFIL Block found in CTPARM.H.
    */

COUNT    OpenISAM(filnam)
         Also- COUNT OpenISAMXtd(filnam)
    ctree(FN_OPNISAM,0,&isamblk)
    /*
    ** isamblk is defined above under the discussion of the
```

```
        ** ISAM block found in CTPARM.H.
        */

    COUNT   PositionSet(keyno,recptr,siglen)
        ctree(FN_MIDSET,keyno,recptr,NULL,NULL,NULL,siglen)

    LONG    PreviousKey(keyno,idxval)
        ctree(FN_PRVKEY,keyno,idxval,&retval)

    COUNT   PreviousInSet(keyno,recptr)
        ctree(FN_PRVSET,keyno,recptr)

    COUNT   PreviousRecord(filno,recptr)
        ctree(FN_PRVREC,filno,recptr)

    COUNT   PutDODA(datno,doda,numfld)
        ctree(FN_PUTDODA,datno,doda,NULL,NULL,NULL,numfld)

    COUNT   ReadData(datno,recbyt,recptr)
        ctree(FN_REDREC,datno,recptr,&recbyt)

    COUNT   ReadVData(datno,recbyt,recptr,bufsiz)
        ctree(FN_RDVREC,datno,recptr,&recbyt,NULL,&bufsiz)

    COUNT   RebuildIFile(ifilptr)
            Also-
    COUNT   RebuildIFileXtd(ifilptr)
        ctree(FN_RBLIFIL,0,&ifilblk)
        /*
        ** ifilblk is defined above under the discussion of the
        ** IFIL Block found in CTPARM.H.
        */

    COUNT   ReleaseData(datno,recbyt)
        ctree(FN_RETREC,datno,NULL,&recbyt)

    COUNT   ReleaseVData(datno,recbyt)
        ctree(FN_RETVREC,datno,NULL,&recbyt)

    COUNT   ReReadRecord(datno,recptr)
        ctree(FN_RRDREC,datno,recptr)

    COUNT   ReReadVRecord(datno,recptr,bufsiz)
        ctree(FN_REDVREC,datno,recptr,NULL,NULL,&bufsiz)

    COUNT   ResetRecord(datno,mode)
        ctree(FN_UPDCURI,datno,NULL,NULL,NULL,NULL,mode)

    COUNT   RestoreSavePoingTRANRST(savpnt)
```

```
    ctree(FN_TRANRST,savpnt)

COUNT   ReWriteRecord(datno,recptr)
    ctree(FN_RWTREC,datno,recptr)

COUNT   ReWriteVRecord(datno,recptr,varlen)
    ctree(FN_RWTVREC,datno,recptr,NULL,NULL,&varlen)

COUNT   SetAlternateSequence(keyno,altseq)
    ctree(FN_SETALTSEQ,keyno,altseq)

COUNT   SetRecord(datno,recbyt,recptr,datlen)
    ctree(FN_SETCURI,datno,recptr,&recbyt,NULL,&datlen)

COUNT   SetSavePoint()
    ctree(FN_TRANSAV,0,&retval2)
    /*
    ** retval2 is a short integer in which the save point number
    ** is returned
    */

COUNT   SetVariableBytes(filno,pvbyte)
    ctree(FN_SETVARBYTS,filno,pvbyte)

COUNT   StopUser()
    ctree(FN_STPUSR)

TEXT *  TransformKey(keyno,tarptr)
    ctree(FN_TFRMKEY,keyno,tarptr)

COUNT   UpdateFileMode(filno,filmod)
    ctree(FN_PUTFIL,filno,NULL,NULL,NULL,NULL,filmod)

COUNT   UpdateResource(datno,resptr,varlen)
    ctree(FN_UPDRES,datno,resptr,NULL,NULL,&varlen)

VRLEN   VDataLength(datno,recbyt)
    ctree(FN_GTVLEN,datno,NULL,&recbyt,NULL,&retval)

VRLEN   VRecordLength(datno)
    ctree(FN_GETVLEN,datno,NULL,NULL,NULL,&retval)

COUNT   WriteData(datno,recbyt,recptr)
    ctree(FN_WRTREC,datno,recptr,&recbyt)

COUNT   WriteVData(datno,recbyt,recptr,varlen)
    ctree(FN_WRTVREC,datno,recptr,&recbyt,NULL,&varlen)
```

Index

All references to the **c-tree Plus Function Reference Guide** are preceeded by the letter **F**.

A

Abort .....4-6, 11-6 - 11-7, F1-3, F1-16
AddKey 4-2, 4-5, 7-5, F1-4, A-3, A-11, A-13
ADDREC
    See AddRecord
AddRecord 3-8, F1-6, F1-88, F1-161
ADDRES
    See AddResource
AddResource .....4-7, 5-2, 5-6, 5-9, 7-8, 8-3 - 8-4,
    .....F1-8, F1-124
ADDVREC
    See AddVRecord
AddVRecord 5-6, 5-9, F1-10, F1-88, A-8
ALCSET
    See AllocateSet
AllocateSet ... 5-8, 7-12, F1-12, F1-15, F1-19, F1-66
ALTER TABLE (sql statement) F4-3
alternate collating sequence 4-7, 5-19, 12-3,
    ..... F1-33, F1-35, F1-67, F1-173, D-6
    See also GetAlternateSequence
    resource 8-6
    See also SetAlternateSequence
Atomicity
    See transaction ,Atomicity
    See Transaction processing
AUTHORIZATION F4-28
automatic recovery .....3-10, 11-7 - 11-8
    See Transaction processing
    See Transaction, automatic recovery
automatic sequence number 5-21

B

B-Tree
    node locks 9-1
    nodes 5-11
BAT_COMPLETE 8-11
BAT_DEL 8-9
BAT_GET 8-9
BAT_LOK_RED 8-10
BAT_LOK_WRT 8-10
BAT_RET_POS 8-10
BAT_RET_REC 8-10
BAT_VERIFY 8-10
Batched operations
    See DoBatch
BATSET
    See DoBatch
Begin 4-6, 11-3, 11-5, F1-13
branching factor 3-5
buffers
    See Index file, buffers
bufs 5-11

C

ChangeSet 5-8, 7-12, F1-12, F1-15, A-10
char
    See TEXT pointers
CHECKLOCK 3-10, 6-1, 10-2, 11-4
CHECKREAD 3-10, 6-1, 10-2
CHGSET
    See ChangeSet
ClearTranError 4-6, 11-7, F1-16, F1-21, F1-176
Client/Server 1-1
CLIFIL
    See CloseIFile
CLISAM
    See CloseISAM
CLOSE (sql statement) F4-4
CloseFile .....4-3 - 4-4, 7-6, F1-17
CloseIFile 5-5, F1-18, A-2
CloseISAM 5-3, 5-5, 7-9, 10-2,
    .....F1-19, F1-66, F1-95, F1-180

CloseRFile 5-5, 8-6, F1-20
CLRFIL
See CloseRFile
CLSFIL
See CloseFile
Collating Sequence
See Alternate Collating Sequence
Commit .. 4-6, 11-3 - 11-5, 11-7, F1-13, F1-16, F1-21
COMMIT (sql statement) F4-5
Common Entry Point .....E-1 - E-12
compiler
Microsoft C 3-3, 9-2
concurrent access 6-1
CONFIG.SYS 3-9
COUNT parameters 3-1
cpybig 3-3
cpybuf ..... 3-3 - 3-4, F1-23
CREATE INDEX (sql statement) F4-6
CREATE SCHEMA (sql statement) F4-7
CREATE TABLE (sql statement) .... F4-8 - F4-10
CREATE VIEW (sql statement) .... F4-11 - F4-12
CreateDataFile 4-2, 4-4, 5-12,
..... F1-24, F1-27, F1-61, A-15
Superfile creation 8-7
CreateDataFileXtd 12-1, F1-27
CreateIFile 5-2, 5-5, 5-16, 7-10, 8-6,
.....F1-19, F1-28, F1-30
CreateIFileXtd 12-1, F1-30
CreateIndexFile ..... 4-2, 4-4, 5-12 - 5-13,
.....F1-31, F1-34 - F1-35, A-15
CreateIndexFileXtd 12-1, F1-34
CreateIndexMember 4-4, 5-12, F1-35
CreateISAM .... 5-1 - 5-2, 5-5, 5-14, 5-16, 7-7, 10-2,
..... F1-19, F1-37, F1-40, A-2, A-7
CreateISAMXtd 12-1, F1-40, F1-181
CREDAT
See CreateDataFile
CREDATX
See CreateDataFileXtd
CREIDX
See CreateIndexFile
CREIDXX
See CreateIndexFileXtd
CREIFIL
See CreateIFile
CREIFILX
See CreateIFileXtd
CREISAM
See CreateISAM
CREISAMX
See CreateISAMXtd
CREMEM
See CreateIndexMember
ctaerr.h 2-1
ctclib.c 9-2, F1-126
ctclib.d 9-2
CTCMPC F1-46
ctcmpl.h F1-6, F1-115, A-3
ctdecl.h 2-1, 3-4, F1-17
cterrc.h 2-1, 3-13
ctexam.p 5-20, 7-14
ctexmc.c 5-20, 7-14
ctexmg.c 5-20, 6-5, 7-14
ctibld.c F1-151
ctifil.h 2-1, 5-25, 7-9
ctixmg.c .....5-9, 5-20, 7-14 - 7-15
ctoptn.h ..... 2-1, 3-4, 5-12 - 5-14, 5-21, 9-1,
..... F1-29, F1-35, F1-38, F1-96, F1-112,
.... F1-126 - F1-127, F1-132, F1-135, F1-150,
..... A-3 - A-4, A-6 - A-8, A-13 - A-14
ctparm.h E-1
ctport.h ..... 3-1 - 3-2, 12-3, F1-4, F1-110, D-3, D-5
ctrbld.c A-7
ctree function E-1
ctsql.h 15-4
ctsrvr.c F1-49, F1-53, F1-155, F1-157,
.....F1-169, F1-189 - F1-190
ctstdr.h 2-1
ctvxam.p 5-20, 5-22, 7-14
ctvxmg.c 5-9, 5-20, 5-22, 7-14
Current ISAM Record F1-174
See ISAM,Current ISAM record
CurrentFileOffset 5-4, 5-6, F1-42
CurrentISAMKey 5-6, 5-24, F1-43

D

d-tree 1-5, 8-6, D-1, D-5
data buffering 11-8
data buffers F1-93, F1-97, F1-137
data file description record 5-11, 7-7
data record
delete flag 3-7
deletion 6-5
first usable F1-114
position 3-13
data types 3-1

DATENT
See NbrOfRecords
datlen 5-12
datno 5-11
DEAD_ERR 6-4
deadlock 9-1
DECADD F1-4
DECLARE CURSOR (sql statement) F4-13
DELBLD
See DeleteKeyBlind
DELCHK
See DeleteKey
DELETE (sql statement) F4-14
deleted records F1-155, F1-157
DeleteFile 4-4, F1-44
DeleteKey ..... 4-3, 4-5, 7-6, F1-45, A-13 - A-14
DeleteKeyBlind 4-3, 4-5, F1-47, A-1, A-11
DeleteRecord . 3-7, 5-3, 5-6, 7-8, 7-13, F1-49, F1-124
DeleteResource .....4-7, 8-3 - 8-4, F1-51
DeleteVRecord 5-6, F1-53
DELFIL
See DeleteFile
Delimiter F1-178, D-3
See Field delimiter
DELREC
See DeleteRecord
DELRES
See DeleteResource
DELVREC
See DeleteVRecord
descending key segment values 5-18
descending key sequence D-6
destbuf (sql function) 15-6, F3-3
destfil (sql function) 15-6, F3-4
DISABLERES 3-10, 8-3
DLOCK 9-2
DLOK_ERR 6-4
DoBatch .....5-8, 8-8 - 8-11, F1-55 - F1-57, A-11
Partial Key Request 8-9
DODA 8-6, F1-146, D-5
DROP TABLE (sql statement) F4-15
DROP VIEW (sql statement) F4-16
Dummy lock file 3-11, 9-2
DUPCHANEL 3-10, 10-3
duplicate key equality search F1-61, F1-74
duplicate keys F1-5, F1-72, F1-107
dynamic memory usage F1-19, F1-37, F1-91,
..... F1-95, F1-134

E

Embedded SQL
See SQL,Embedded
ENABLE 6-2, 6-4, F1-110
ENABLE_BLK 6-2, 6-4, F1-110
EnableResource 4-7, 8-3, 8-5, F1-59
ENARES
See EnableResource
EQLKEY
See GetKey
EQLREC
See GetRecord
err_pos 14-3
error code 3-13
Error handling 5-3
EstimateKeySpan 4-5, F1-60
ESTKEY
See EstimateKeySpan
exact match of duplicate keys F1-61, F1-74
EXCLUSIVE 3-10, 6-1
extended file functions F1-27, F1-30, F1-34,
..... F1-40, F1-93, F1-97, F1-129 -
.....F1-130, F1-133, F1-137, F1-153
extended function calls
file extensions 5-25
superfiles 8-7
Extended functions 4-3, 5-4, 12-1, F1-181

F

FairCom Server
See Server
FairCom SQL Server
See Server,SQL
FETCH 15-5
FETCH (sql statement) 15-6, F4-17
FETCHP (sql statement) F4-18
Field delimiter 3-9, 5-15, F1-178
field type D-3
File
creation ..... F1-24, F1-27 - F1-28, F1-31,
..... F1-34, F1-37
deletion F1-44
numbering 3-11
File mode 3-8, 3-11, 11-3, 11-5, F1-24
DISABLERES 8-3

multi user 6-1
SUPERFILE 8-7
file numbering 10-4
File passwords
See Security
File Permissions
See Security
File recovery 3-13
See also RebuildIFile
File Server ..... 10-1 - 10-4
filmod ..... 5-12 - 5-13, 6-1
See file mode
filnam ..... 5-11 - 5-12
first usable data record F1-114
FirstInSet ..... 5-8 - 5-9, 7-12, F1-12, ... F1-15, F1-61, F1-74, F1-118, F1-141, A-7
FirstKey 3-12, 4-5, F1-63, A-13
FirstRecord 3-12, 5-6, 7-11, F1-64
FIXED 3-9
fixed record length
See Record,Fixed length
FPUTFGET 9-1, F1-91, F1-108, F1-123
FRCKEY
See KeyAtPercentile
FREE ..... 6-3 - 6-4, F1-110
FreeSet 5-8, F1-66
FRESET
See FreeSet
FRSKEY
See FirstKey
FRSREC
See FirstRecord
FRSSET
See FirstInSet

G

GetAlternateSequence 4-7, 5-19, F1-67
See also alternate collating sequence
GETALTSEQ
See GetAlternateSequence
GETCURK
See CurrentISAMKey
GETCURP
See CurrentFileOffset
GetDODA 4-7, 8-6, F1-68
GETFIL
See GetFileInfo
GetFileInfo 4-4, F1-43, F1-70
GetGTEKey 4-6, 5-9, F1-72
GetGTERecord F1-74
GetGTKey 4-6, 5-9, F1-76
GetGTRecord 5-6, 7-13
GetIFile 5-5, 8-6, F1-77
GetKey ..... 4-2 - 4-3, 4-5, 7-5, F1-72, F1-78
GetLTEKey 4-6, F1-79
GetLTERecord 5-6, F1-80
GetLTKey 4-6, 5-9, F1-82
GETNAM
See GetSymbolicNames
GetRecord 5-2, 5-6, 7-8, F1-74, F1-83
GETRES
See GetResource
GetResource ..... 4-7, 8-3 - 8-4, F1-85
GetSerialNbr F1-88
GetSymbolicNames 4-4, F1-89
GetTempFileName F1-90
GETVLEN
See VRecordLength
Group ID
See Security
GTEKEY
See GetGTEKey
GTEREC
See GetGTERecord
GTKEY
See GetGTKey
GTVLEN
See VDataLength
GUEST
See Security

H

Header record 3-13

I

I/O Management 10-3
IDXENT
See NbrOfKeyEntries
IFIL structure 3-2, 5-25, 7-9
IGRANT (sql statement) ..... F4-19 - F4-21
IIDX Structure 5-26, 7-10
INCADD F1-4
Include statements 2-1

Incremental ISAM F1-18, F1-20, F1-28,
..... F1-37 - F1-38, F1-95, F1-127,
..... F1-131, F1-134 - F1-135, F1-150
creation F1-30
include 5-2
See ISAM, Incremental ISAM structure
Resources 5-29
Index file
buffers 3-7, 5-11, 7-3
description record 5-12, 7-7
index member 3-11, F1-31
index member record 5-14
InitCTree 3-12, 4-1, 4-4, 7-3, 10-2, 15-1,
..... F1-31, F1-91, F1-93, A-2 - A-3, A-7
InitCTreeXtd 12-1, F1-93
initialization record 5-11, 7-7
InitISAM ... 5-5, 7-10, 10-2, F1-95, F1-97, A-2, A-7
InitISAMXtd 12-1, F1-97, F1-181
INSERT INTO (sql statement) ..... F4-22 - F4-23
Integer
four byte 3-2
signed 3-1
unsigned 3-2
INTISAM
See InitISAM
INTISAMX
See InitISAMXtd
INTREE
See InitCTree
INTREEX
See InitCTreeXtd
Isam
record rewrite 5-3
creation F1-40
Current ISAM record 5-3, 7-13
Data record locks 6-2
Errors 5-3
file relationships 5-9
functions .....3-12, 5-1 - 5-29
Incremental 2-1
Incremental ISAM 5-5
Incremental ISAM structure 3-6, 5-8,
.....5-24 - 5-29, 7-9, 7-14 - 7-15
Incremental ISAM, resources 8-6
logging off server 10-2
logging on to Server 10-2
parameter file ..... 3-6, 5-10 - 5-15, 7-6
Parameters .....5-1, 5-8 - 5-9
ISAM, Current ISAM record F1-158, F1-165,
.....F1-168, F1-170
isam.c 5-1
isam_err 3-13, 5-3, 7-8, F1-1
isam_fil 3-13, 5-3, 7-8, F1-1
ISEG Structure 5-27, 7-10
ITIM_ERR 6-4

K

Key
key value padding 7-4
translation 7-4
key segment
parameter 5-1, 5-9
descending values 5-18
description record 5-14, 7-8
key value padding F1-5, F1-107
key value sets F1-12, F1-61
key value suffix F1-5, F1-107
KeyAtPercentile 4-5, F1-60, F1-99
keydup 5-12, 5-14
keylen 5-12, 5-14
keyno 5-12, 5-14
Keys ..... 3-4 - 3-6
compressed ..... 3-4 - 3-6
duplicate 3-6
fixed length ..... 3-4 - 3-5
ISAM 3-6
key length 3-6
key segment parameter 3-12
key types 3-4
key value assembly 5-23
key value padding 3-4, 4-2, 5-9
key value suffix 3-6
segment length 5-15
segment mode ..... 5-16 - 5-23, D-1
segment position 5-15
sequence number 5-18
target key value 5-24
keytyp 5-12, 5-14

L

LAN 1-2, F1-152
LastInSet .....5-8 - 5-9, F1-12, F1-15, F1-80,
..... F1-101, A-7
LastKey 4-6, F1-103, A-13

LastRecord 5-6, F1-104
Resources 8-3
LKISAM
See LockISAM
LoadKey 4-6, F1-106, A-4
local file buffering 10-1
LOCK 9-2
LockData ..... 4-5, 6-3 - 6-4, 7-5, 11-4,
.....F1-21, F1-86, F1-108, F1-111
Locking
File Locking ..... 6-1 - 6-3
ISAM data record locks 6-2
See Record locking
LockISAM .....5-6, 6-2, 7-8, 8-10, 11-4 - 11-5,
..... F1-13, F1-21, F1-86, F1-110, A-7
transactions 11-3
Logging off Server 4-4, 5-5, 10-2, F1-180
Logging on Server 4-4, 5-5, 10-2
LOKREC
See LockData
LONG values 3-2
Low Level Functions .....3-11, 4-1 - 4-7
lowlevel.c 4-1
LSTKEY
See LastKey
LSTREC
See LastRecord
LSTSET
See LastInSet
LTEKEY
See GetLTEKey
LTEREC
See GetLTERecord
LTKEY
See GetLTKey

M

MAX_DAT_KEY F1-29, F1-38,
.....F1-128, F1-132, F1-135, F1-151
MAX_KEY_SEG F1-29, F1-38, F1-128,
.....F1-132, F1-135, F1-151
MAX_LOCKS F1-112
MAXFIL F1-38, F1-96, F1-135
MAXLEN ..... F1-35, F1-38, F1-126 - F1-127,
.....F1-132, F1-135, F1-150
Members F1-35
Microsoft C
See Compiler, Microsoft C
MIDSET
See PositionSet
minimum record length 3-9
Multi user 1-3, 3-10, 7-3, 9-1
interference 6-4
multi tasking 6-1, 10-1
multi-tasking 3-10
updates 6-5
Multiple indices 7-14
multiple sets F1-12

N

NbrOfKeyEntries 4-6, F1-123
NbrOfRecords 4-5, F1-124
Network 3-10, 6-1, 10-1
See also LAN
NewData ...3-8, 4-2, 4-5, 7-5, F1-88, F1-114, F1-124
NEWREC
See NewData
NewVData 4-5, F1-88, F1-116, F1-157
NEWVREC
See NewVData
NextInSet 5-8, 7-12, F1-12, F1-15, F1-118, A-7
NextKey 3-7, 4-6, F1-120
NextRecord 5-6, 7-11, 7-13, F1-121
node locks 9-1
node sectors 5-11
nomemb 5-12
null
character 3-3
null byte 3-4, 5-9
null key flag 5-13, 7-7
NXTKEY
See NextKey
NXTREC
See NextRecord
NXTSET
See NextInSet

O

OPEN (sql statement) F4-24
OpenFile 3-12, 4-2, 4-4, 5-11,
..... F1-31, F1-125, F1-130, A-15
Superfile 8-7

OpenFileWithResource ..... 5-2, 5-5, 8-6, 12-1, ... F1-127, F1-129
OpenFileWithResources F1-28
OpenFileWithResourceXtd F1-129
OpenFileXtd 12-1, F1-130
OpenIFile .....5-2, 5-5, 5-11, 5-16, 7-10, F1-19, ... F1-131, F1-133, A-2, A-7 - A-8
OpenIFileXtd 12-1, F1-133
OpenISAM .....3-12, 5-1 - 5-2, 5-5, 5-14, 5-16, 7-7, ... 10-2, F1-19, F1-134, F1-137, A-2, A-7
OpenISAMXtd 12-1, F1-137, F1-181
OPNFIL
See OpenFile
OPNFILX
See OpenFileXtd
OPNIFILX
See OpenIFileXtd
OPNISAM
See OpenISAM
OPNISAMX
See OpenISAMXtd
OPNRFIL
See OpenFileWithResource
OPNRFILX
See OpenFileWithResourceXtd

P

Padding 5-16, F1-178
Set padding value 3-5
Partial Key Request 8-9
Passwords
See Security
PCNET 7-1
Performance
buffering 3-10, 11-8
disk head movement 3-5
multiple channels 10-3
PERMANENT 3-8
Permanent file open 3-8
Permission mask
See Security
Permissions
See Security
PKEYREQ 8-9, F1-55
PositionSet 5-8, F1-139
PREIMG ..... 3-10, 10-2, 11-4 - 11-5
preprocessor
See SQL,preprocessor
PreviousInSet 5-8, F1-12, F1-15, F1-141, A-7
PreviousKey 4-6, F1-143
PreviousRecord 5-6, 7-13, F1-144
Resources 8-3
productivity
message traffic 10-1
PROTOTYPE 3-4
Prototyping 3-3
PRVKEY
See PreviousKey
PRVREC
See PreviousRecord
PRVSET
See PreviousInSet
PutDODA 4-7, 8-6, F1-146, D-1, D-5
PUTFIL
See UpdateFileMode

R

r-tree ..... 1-5, 5-11 - 5-14, 5-21, 7-7, ... F1-89, D-5 - D-6
DODA 5-13
symbolic index name 5-13
r_fetch (sql function) F3-5
r_insert (sql function) F3-6
r_update (SQL function) F3-7
RBLIFILX
See RebuildIFileXtd
RDVREC
See ReadVData
read only access 6-1
ReadData 3-12, 4-3, 4-5, 7-6, F1-147, A-3
READFIL 3-10, 6-1
READREC ..... 6-2 - 6-4, F1-110
ReadVData 4-5, F1-148
Rebuild .....5-12 - 5-13, 7-6, 7-14 - 7-15
function F1-150, F1-153
RebuildIFile 3-13, 5-5, F1-150, F1-153
RebuildIFileXtd 12-1, F1-153
Record
deletion 6-5
fixed length 3-7, 3-9
minimum length 3-9
record length 3-7
relative record number 3-7

rewrite 6-5
update 6-5
Record locking ..... 6-1 - 6-3
Record Schema .....3-9, 5-15, F1-146, D-1 - D-2
record schemas 12-3
REDREC
See ReadData
REDVREC
See ReReadVRecord
REGADD F1-4
Relational database 13-1
ReleaseData 3-7, 4-3, 4-5, 7-6,
.....F1-114, F1-124, F1-155
ReleaseVData 4-5, F1-157
ReReadRecord 5-7, F1-158, A-3
ReReadVRecord 5-6, F1-61, F1-64, F1-74,
.....F1-80, F1-83, F1-101, F1-104, F1-118,
..... F1-121, F1-141, F1-159, A-3, A-8 - A-9
RESET 6-3, F1-110
ResetRecord 5-4, 5-7, F1-161
Resource ..4-7, 8-1 - 8-5, F1-85, F1-104, F1-144, D-4
Alternate Collating Sequence ..F1-67, F1-173
disable 8-3
DODA 4-7
Enable after creation F1-59
file mode 3-8
Functions 8-3
IFILE open F1-127
IFIL structure F1-28
Incremental ISAM 5-5, 5-29, 8-6
Incremental ISAM close F1-20
LastRecord 8-3
locking 6-3
mode 3-10
Name 8-2
Number 8-2
PreviousRecord 8-3
record marker 3-8
record schema F1-68, F1-146, D-1
Type 8-2
RESTORE 6-3, F1-110
RESTORE_BLK 6-3, F1-110
RestoreSavePoint .....4-6, 11-6 - 11-7,
.....F1-16, F1-21, F1-163
RESTRED 6-3, F1-110
RETREC
See ReleaseData
RETVREC
See ReleaseVData
reusing deleted records F1-155, F1-157
REVOKE (sql statement) F4-25
rewrite
See record,rewrite
ReWriteRecord 5-3, 5-7, 5-18, 7-13,
..... F1-161, F1-165, A-1
ReWriteVRecord 5-3, 5-7, F1-168, A-1
RLOCK 9-2
ROLLBACK WORK (sql statement) F4-26
Rows
See SQL,Rows
RRDREC
See ReReadRecord
RTREE A-6
RWTREC
See ReWriteRecord
RWTVREC
See ReWriteVRecord

S

savepoints
See Transaction,savepoints
sect 5-11
security ..... 4-3 - 4-4, 5-4, 12-1 - 12-4
File passwords 12-4
File Permissions 12-2
function F1-171
Groups 12-2
GUEST group 12-2
modify permissions F1-171
Passwords 12-2
Permission mask 12-3
permission masks F1-27, F1-30,
.....F1-34, F1-40, F1-153
resource 8-6
User ID 12-2, F1-40, F1-93, F1-97, F1-137
user password F1-40, F1-93, F1-97, F1-137
user profile mask .F1-40, F1-93, F1-97, F1-137
SELECT (sql statement) F4-27
SELECT INTO 15-5
Sequence number 3-7
serial number ..5-18, F1-88, F1-114, F1-116, F1-168
See also GetSerialNbr
Serializability, ANSI feature 14-3
SERIALNUM
See GetSerialNbr
Server .....1-1 - 1-2, 1-5, 3-8, 3-10 - 3-11, F1-32

logoff 4-4, 5-5, 10-2, F1-180
logon 4-4, 5-5, 10-2
security 12-1
server name F1-40, F1-93, F1-97, F1-137
server-application relationship 10-1
SQL .....1-1, 1-5, 13-1 - 13-2, D-1
system administrator 12-2
SET AUTHORIZATION (sql statement) ...F4-28
Set Functions 5-7, 7-11, F1-12, F1-15, F1-61,
.....F1-66, F1-101, F1-118, F1-139, F1-141
SetAlternateSequence 4-7, 5-19, F1-33, F1-35,
..... F1-173, D-7
See also alternate collating sequence
SETALTSEQ
See SetAlternateSequence
SETCURI
See SetRecord
setdest (sql function) 15-6, F3-8
SetRecord 5-4, 5-7, F1-174
sets of key values F1-12, F1-61
SetSavePoint .....4-6, 11-6 - 11-7, F1-176
SETVARBYTS
See SetVariableBytes
SetVariableBytes 3-9, 4-7, 5-15, 8-6,
.....F1-178, D-3, D-6
setvars 15-5
setvars (sql) F3-9
SHARED 3-10, 6-1, 7-3
single entry point F1-181
Single user mode 7-3
Non-Server 1-3
Server 1-3
SORT_SPACE F1-151
SQL
Columns 14-1
Embedded ..... 1-3, 13-1, 15-1 - 15-3, F3-1
embedded SQL 13-1
Error codes .....14-3 - 14-4, B-1 - B-10
interactive 13-1
interactive SQL 13-1
preprocessor 15-1
record schema D-1
rows 14-1
Select statement ..... F2-1 - F2-22
See also Server,SQL
Tables 14-1
view 14-1
Views 14-1
SQL preprocessor ..... F4-17 - F4-18
sql1 (sql function) 15-5, F3-10
sql2 (sql function) 15-5, F3-11
sql_err 14-3
StopUser 4-4, 7-4, 7-9, 15-1, F1-180
STPUSR
See StopUser
suffix F1-5, F1-107
Superfile .....3-10, 8-6 - 8-7, F1-25, F1-32,
.....F1-64, F1-104, F1-121, F1-125, F1-144
file mode 3-8
Members 8-7
NewData F1-114
SQL 14-1
SUSPEND 6-3, F1-110
SWTCURI 5-4
symbolic data field name 5-12
symbolic index name 5-13
System Administrator 12-4

T

Tables
See SQL,Tables
target key value F1-181
Temporary File Name F1-90
TEXT pointers 3-2
TFRMKEY
See TransformKey
TMPNAME
See GetTempFileName
TRANABT
See Abort
TRANBEG
See Begin
TRANCLR
See ClearTranError
TRANEND
See Commit
TRANRST
See RestoreSavePoint
Transaction 1-3, 3-8, 3-10
abort 11-6
Atomicity 11-2
automatic recovery 11-2
commit 11-5
locking F1-111
mode 10-2
number 11-5

processing . 5-12, 10-3, 11-1 - 11-8, F1-6, F1-10
savepoints 11-6
SQL transaction processing 14-2
TRANSAV
See SetSavePoint
TransformKey .....5-7, 5-18, 5-24, 7-8, 7-12 - 7-13, ...F1-40, F1-93, F1-97, F1-137, F1-181, E-2
TRNLOG 3-10, 10-2, 11-3, 11-8
TRNLOG ..... 11-4 - 11-5
two-phase locking protocol 6-2, F1-108, F1-111

U

UCOUNT parameters 3-2
UDLOCK 9-2
uerr_cod 3-13, 4-3, 7-4, 14-4, F1-1
UINT parameters 3-2
UNIX 1-2
UNLOCK 9-2
UPDATE (sql statement) F4-29
update record 6-5
UpdateFileMode 4-4, F1-183
UpdateResource .....4-7, 8-3 - 8-4, F1-185
UPDCURI
See ResetRecord
UPDRES
See UpdateResource
User ID
See Security
user profile mask . 5-18, F1-40, F1-93, F1-97, F1-137
USERPRF_NTKEY ...F1-40, F1-93, F1-97, F1-137

V

variable length
data records 5-15, 7-14
records 3-9
variable length record .....F1-116, F1-148, F1-157, ... F1-159, F1-168, F1-187 - F1-188, F1-190
key segment 5-18
variable length records
batch processing F1-57
varying length field 5-10
VDataLength F1-148, F1-187
Views
See SQL,Views
VIRTUAL 3-8
Virtual file open 3-8
VLENGTH 3-9
VOID 3-2
VRecordLength 4-5, 5-6, F1-159, F1-188
VRLEN parameters 3-2

W

white space 5-11
WriteData 4-2, 4-5, 7-5, F1-189, A-3
WRITETHRU 3-10, 11-8
WriteVData 4-5, F1-190, A-3
WRTREC
See WriteData
WRTVREC
See WriteVData

X

XENIX
See UNIX
xtdsiz .....5-12 - 5-13, 7-3

Z

zero byte 3-4

Reader's Review

The **FairCom** Documentation Department wants to provide manuals that meet the needs of all of our users. After reading and using this manual, please take a few moments to answer the following questions. Your input can help us more fully meet your needs in the development of future documentation. Your name, company, and phone number are optional, but will help us to serve you better.

Manual Name	*Your Company Name*
Operating Environment you are programming in	*Your Telephone Number*
Your Name	*Type of products you are developing*

Please rate this manual in the following areas:	*10 = Highest 0 = Lowest*
Clarity of instructions	
Completeness of information	
Logical organization	
Easy to read	
Ease of finding information	
Usefulness of examples	
Usefulness of reference material *(Table of Contents, Index, Appendices)*	

Please continue on the back of this page

Are there areas of the manual that you found especially helpful? If so, what are they?

Are there other areas of the manual that you feel could be improved? If so, how?

Are there any particular concepts that show be explained in more detail?

Was there any specific information that you had a difficult time finding?

Please send these or any other comments/suggestions to the following address:

FairCom Corporation
Documentation Department
4006 West Broadway
Columbia, MO 65203

NOTES

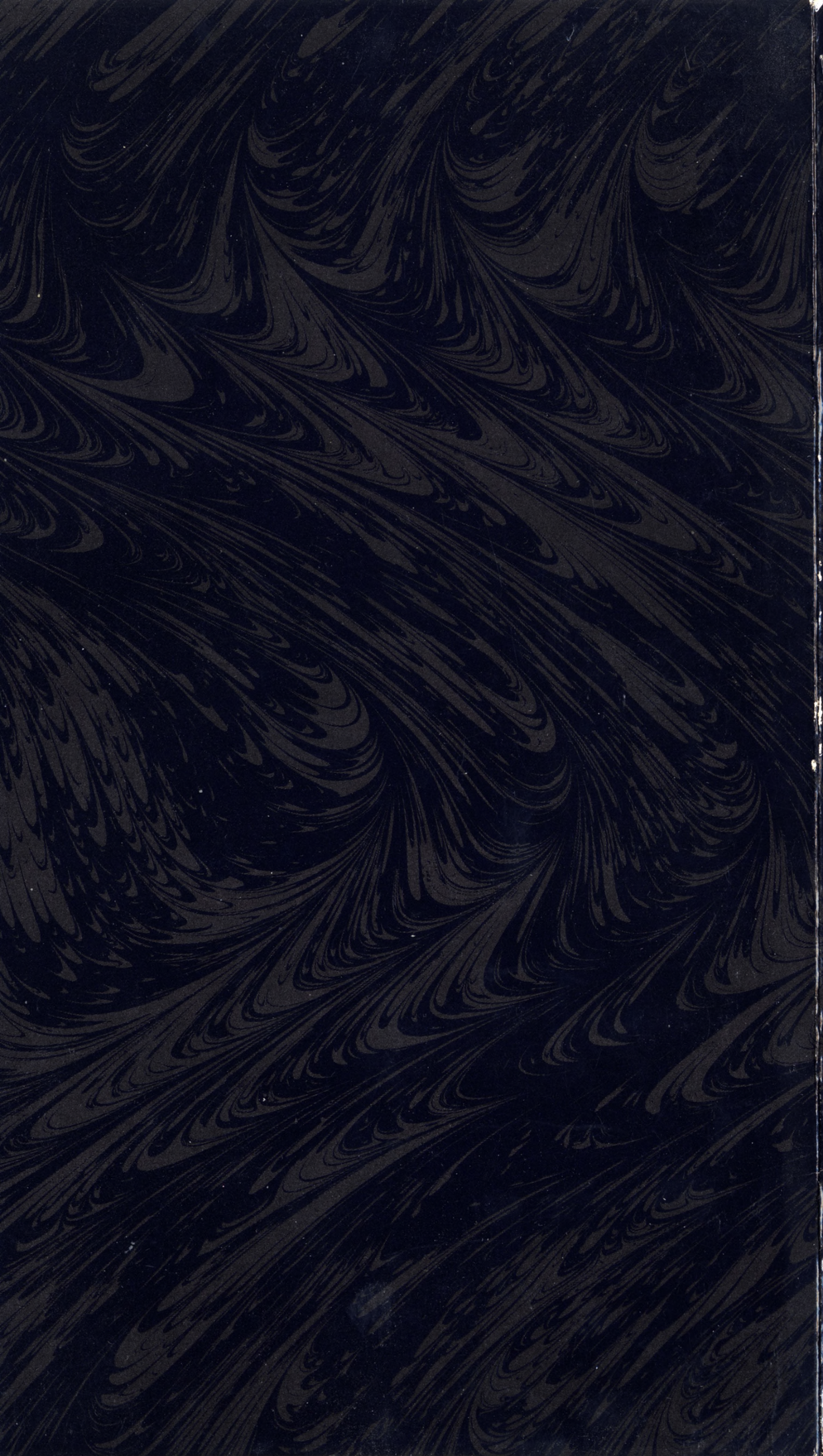
NOTES

NOTES

NOTES

c-tree Plus™

Programmer's Reference Guide



FAIRCOM corporation

4006 WEST BROADWAY
COLUMBIA, MO 65203
PHONE 314.445.6833
800.234.8180
FAX 314.445.9698